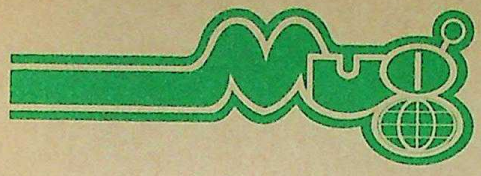

REVISED 1979

Mumps Primer

R. F. Walters
J. Bowie
J. C. Wilcox

MUMPS PRIMER, REVISED

R. F. Walters
J. Bowie
J.C. Wilcox

MUMPS Development Committee

1979

PROPERTY OF NAVAL WEAPONS CENTER

Copyright March 1979 by the MUMPS Users' Group, Box 208, Bedford MA 01730

ISBN 0-918118-18-2

315
D12

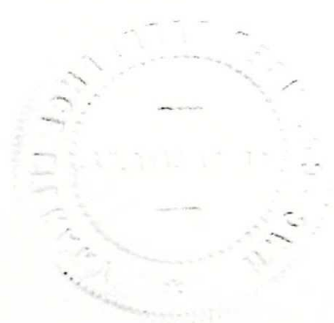

Table of Contents

	Page
Chapter 1 - Introduction to MUMPS	1
What is MUMPS	1
A Brief History	1
How to Use This PRIMER	3
For the Experienced Programmer	4
Where to Go for Further Information	4
Acknowledgements	5
References	5
Chapter 2 - Basic Components of MUMPS	7
Character Set	7
Numbers	7
Numeric Ranges and Precision	10
String Literals	10
Variables	11
Functions	12
Special Variables	12
Summary	12
Chapter 3 - Operations on Arithmetic Values	15
Binary Operators	15
Arithmetic Expressions	17
Unary Operators	19
Numeric Interpretation	19
Arithmetic Function: $RANDOM	20
Summary	21
Chapter 4 - Logical Operations, Comparisons	23
Arithmetic and Relational Operators	23
Logical Operators	25
Summary	26
Chapter 5 - Operations on Strings	29
String Operators:	29
String Functions	30
$LENGTH	30
$FIND	31
$EXTRACT	32
$PIECE	33
$ASCII	34

Page

$CHAR 35
Summary 35

Chapter 6 - Introduction to Commands 37

What is a Command? 37
SET Command 38
KILL Command 39
$SELECT Function 40
WRITE Command 41
$X and $Y Special Variables 42
READ Command 42
Postconditionals on Commands 43
Abbreviated Forms of Commands 44
Summary 45

Chapter 7 - Command Lines 47

Format of a Command Line 47
IF Command and $T Special Variable 49
ELSE Command 50
Summary 51

Chapter 8 - Routine Structure 53

GOTO Command 53
DO and QUIT Commands 54
Posconditionals on Arguments 56
HALT and HANG Commands 56
Summary 57

Chapter 9 - Program Structure 59

GOTO and DO/QUIT Commands with Routines 59
Packages 62
Summary 63

Chapter 10 - Local Arrays 65

Introduction 65
FOR Command 66
GOTO and QUIT in FOR 70
More About Arrays 71
Multiple Subscripts 71
Permissible Subscript Values 72
Placement of Data Values 72
KILL as Used with Arrays 73

Page

Array Manipulation Functions: 73
$DATA 73
$NEXT 74
Summary 75

Chapter 11 - Advanced Input/Output 77

OPEN Command 77
USE Command 78
CLOSE Command 79
Extensions of READ and WRITE 80
$JUSTIFY Function 81
Summary 82

Chapter 12 - Global Variables 83

Naked References 84
LOCK Command 86
Summary 89

Chapter 13 - Some Special Features of MUMPS 91

Pattern Match 91
$HOROLOG Special Variable 94
$TEXT Function 96
Summary 97

Chapter 14 - Indirection 99

Summary 102

Chapter 15 - Implementation-Specific Components of MUMPS 103

BREAK and VIEW Commands, $VIEW Function 103
$JOB and $STORAGE Special Variables 104
Implementation Defined Language Elements 104
Direct Mode 105
Summary 106

Chapter 16 - The Future of MUMPS 107

The Process of Change 107
New Language Features 108
Limitations to Language Evolution 109
Portability as an Aid to Evolution 110
Effects of Technological Advances 110

Page

Summary 111

Chapter 17 - An Applications Example - Keyword-in-Context 113

Introduction 113
Design of KWIC Package 114
System Driver Routine 114
Data Entry 115
Generation of KWIC Index 117
Application Example 119
Summary 122

Chapter 18 - Another Applications Example - Mailing List Package 123

Introduction 123
Design of the Pakcage 123
Description of Routines 125
Listing of Routines 128
Sample of Output Using Mail List Package 132
Summary 137
Acknowledgements 137

Chapter 19 - Standard MUMPS Guide *139*

Index to Language Features

	Page
Commands	
BREAK	103
CLOSE	79
DO	54,59
ESE	49
FOR	66
GOTO	53,59,70
HALT	56
HANG	56
IF	49
KILL	39,73
LOCK	86
OPEN	77
QUIT	54,70
READ	42,80
SET	38
USE	78
VIEW	103
WRITE	41,80
XECUTE	99
Z-Commands	104
Functions	
$ASCII	34
$CHAR	35
$DATA	73
$EXTRACT	32
$FIND	32
$JUSTIFY	81
$LENGTH	30
$NEXT	74
$PIECE	33
$RANDOM	20
$SELECT	40
$TEXT	96
$VIEW	103
$Z Functions	104
Indirection	99

Page

Operators

Arithmetic - Binary 15
Arithmetic - Relational 23
Arithmetic - Unary 19
Logical 25
Pattern Match 91
String 29

Postconditionals on Arguments 56
Postconditionals on Commands 43

Special Variables

$HOROLOG 94
$IO 79
$JOB 104
$STORAGE 104
$TEST 49
$X 42
$Y 42
$Z 104

Timeouts 77

Chapter 1

Introduction

What is MUMPS?

The term MUMPS has been used to define a number of concepts related to computers. The acronym MUMPS stands for Massachusetts General Hospital's Utility MultiProgramming System, but in the years since its conception in 1967, the term has taken on a variety of meanings.

MUMPS may be considered to embody three concepts. First, it is an interpreted computer language. "Interpreted" because it may be executed directly without conversion to "object code" as required by compiled languages. In addition, MUMPS, because of its ability to handle large dynamic files (such as medical data bases), may be considered to be a file management system whose capabilities rival those of many languages ("data base management systems") designed solely for that purpose. Finally, since MUMPS has traditionally been implemented on small computers as a single language system, the term has been used to include the concept of an operating system. This latter component of the language is not inherent in the definition of MUMPS, though the interpreter and file management aspects are essential components.

The brief history of the language provided in the next section offers some explanation of the diversity of viewpoints regarding MUMPS. Whatever its background, MUMPS is now one of four computer languages accepted by the American National Standards Institute (ANSI). It has been implemented on a variety of computers, ranging from microprocessors through minicomputers to large mainframes. It is well known in Europe and Japan, and its users include a surprising predominance of commercial installations.

A Brief History of MUMPS

The concept of interactive computing on timeshared systems had its beginnings in the early 1960s. In late 1967, the Laboratory of Computer Science at Massachusetts General Hospital, recognizing the merits of interactive systems in the medical field, began development of a language that incorporated the ability to manipulate text strings with a file management system in an interactive timeshared environment operating on minicomputers. The first publication describing the

language was presented at the 1969 Fall Joint Computer Conference (1). Because of the success achieved by its designers in meeting their objectives, the language soon gained popularity in clinical applications and also in medical education. Its fame spread beyond the confines of the founding institution, and soon a number of dialects sprang up, each modifying or expanding on the original MUMPS in ways intended to improve on the original design. New applications were developed, some in the original language, others in the various dialects available through private or commercial sources. A considerable amount of confusion resulted, culminating, by 1972, in the existence of no less than 14 separate operational or developmental versions of the MUMPS language.

Because of the potential of the language, a cooperative effort, jointly sponsored by the National Bureau of Standards and the National Center for Health Services Research (a branch of the Department of Health, Education and Welfare), was initiated in 1972 to foster the development of a new standard MUMPS language. This effort involved the principal implementors and users of the extant MUMPS dialects. It was conceived as a project by consensus, not by fiat. The MUMPS Development Committee, formed to spearhead the language definition, adopted a formal constitution and by-laws to ensure impartial consideration of all viewpoints. Meetings were held several times a year from 1973 to 1975, by which time a new Standard MUMPS language had been agreed upon and adopted. This language definition was submitted to ANSI for approval in the spring of 1976, and, following the canvass method prescribed by ANSI, the language was approved without dissent by ANSI in September, 1977.

Since that time, the language has been implemented on a variety of systems. Standard MUMPS is now available on microcomputers, minicomputers, and even on some large mainframe systems. Its use is increasing in medical, commercial and educational environments. Interest in the language is also increasing, with the likely result that still more implementations will be announced in the months and years directly ahead.

The MUMPS Development Committee is continuing its responsibility for language review and evolution. Meeting at least twice a year, they consider possible language extensions or revisions. Those changes that are formally adopted by the Development Committee will be proposed as modifications in the mandatory 1982 language update specified by ANSI regulations. In the meantime, the Development Committee and its application-oriented counterpart, the MUMPS Users' Group, are continuing to refine documentation describing the language and its uses. Persons interested in further information are urged to contact the representative listed below.

How to Use This Primer

This edition of the PRIMER (the second to be published by the MUMPS Development Committee) is intended to serve the beginning student of the language. Some familiarity with programming concepts will be helpful; however, the PRIMER may serve as a supplementary text in introductory programming language classes in which MUMPS is used to teach those concepts. For this reason, the PRIMER is organized in such a way as to introduce basic concepts with simple examples before moving to those characteristics that distinguish MUMPS from some of the other computer languages familiar to the experienced programmer.

In spite of this orientation, however, the PRIMER may serve a useful purpose for individuals with previous experience. Some suggestions for selective browsing by advnced programmers are provided in the next section of this chapter.

The beginning student will want to use the PRIMER in conjunction with a computer on which Standard MUMPS has been implemented. Ideally, the novice should proceed through the text and examples in short sessions interspersed with sessions with the actual language. Tutorial assistance from co-workers or instructors will be helpful in small doses, but the main learning experience can only be achieved through active participation, wrestling with difficult concepts before asking for help. Review of language concepts already covered will be necessary for almost all users, since the retention curve of this material is aided by reinforcement.

At some point, it may be useful for the student to consider a real application that he has conceived in his own area of interest. Using this application as a frame of reference will probably be helpful in gaining a firmer understanding of many of the language constructs that may appear easy in the abstract but offer additional learning problems when considered in the specific.

In order to aid both the beginning and more experienced programmer in the use of this text as well as the MUMPS language itself, a companion reference is included as an insert. This reference, the Standard MUMPS Pocket Guide, summarizes the principal features of the languge in a handy reference format that will save time in actual progrmming sessions.

Two further aids are provided to make this PRIMER additionally valuable. A chapter outlining a typical application is provided to give the user a better feeling for the design and implementation considerations adopted by experienced users facing a real world situation. This example may we studied at several different levels as

the student moves through the earlier programming concepts. Finally, a concluding chapter has been left blank, to be completed by the manufacturer or implementor of the system which you are using. For those implementors who have provided features not specified in the Standard, this chapter may be included to help the user take full advantage of such capabilities.

For the Experienced Programmer Who Wants to Learn About MUMPS

Persons who have programmed computers in other languages will find many parts of the PRIMER oversimplified. However, there are some sections that may be beneficial to some users whose experience does not include familiarity with concepts embedded in Standard MUMPS.

Chapter 2 contains descriptions of string literals and dynamic variables unlike those used in FORTRAN and COBOL, but somewhat similar to certain dialects of BASIC. Arithmetic precedence, described in Chapter 3, is unlike FORTRAN and should be understood to avoid computational errors.

One of the principal features of MUMPS is its ability to manipulate textual strings. These operations are described in Chapters 5 and 13. Arrays in MUMPS, being both dynamic and sparse, require particular attention; Chapter 10 describes local subscripted arrays, whereas the concept of Global variables is discussed in Chapter 12.

Indirection, a language feature that permits self-modifying code, is unfamiliar to progrmmers in most other languages. This concept is introduced in Chapter 14.

Future considerations of the MUMPS language are discussed in Chapter 16. Finally, Chapters 17 and 18 give examples of two applications that illustrate many typical MUMPS programming techniques.

Where to Go for Further Information

Several key references to the MUMPS language and its uses are available through the MUMPs Users' Group. The ANSI Standard MUMPS (X 11.1) and Programmers's Reference Manual are perhaps the most important of these publications for serious users of the language. Information on obtaining these and other documents may be obtained by writing

Richard Zapolin, Executive Director
MUMPS Users Group
P.O. Box 204
Bedford, MA 02114

Persons interested in comparing the MUMPS language with other programming systems will find the article by Munnecke, et al., helpful (2). For those wishing to move to more advanced programming techniques, the publication by Beckley and Bridger (3) is a valuable aid.

A tutorial program aimed at assisting the novice in MUMPS has been developed by Zimmerman, et al., (4). This program can also be obtained through the MUG Executive Director.

Acknowledgments

The first edition of the PRIMER was prepared by Martin E. Johnson and Ruth E. Dayhoff. Although the current version represents a substantial revision, its authors have retained many passages from the original edition, for which they express their thanks.

Drafts of this revised edition have been reviewed by numerous members of the MUMPS community, many of whom have provided valuable suggestions incorporated in the final text. Among the major contributors to this process are Martin Johnson, Richard Zapolin, Henry Heffernan, Stephen Johnson, and Arden Forrey. The authors acknowledge their assistance with gratitude, and accept full responsibility for any inaccuracies or confusing passages that may remain despite their valuable efforts.

References

1. Greenes, R.A., A.N. Pappalardo, C.W. Marble and G.O. Barnett (1969), "Design and Implementation of a Clinical Data Management System," Computers in Biomedical Research, V. 2, pp 469-485.

2. Munnecke, T., R.F. Walters, J. Bowie, C.B. Lazarus and D.A. Bridger (1977), "MUMPS: Characteristics and Comparisons with Other Programming Systems," Medical Informatics, V. 2, pp. 173-196.

3. Beckley,R. (1977)," Advanced Programming Techniques", MUMPS Users' Group, St. Louis, Mo.

4. Brigham, C.R., J.D. Halvorson and J. Zimmerman (1976), "QUEST: A Teaching Program Driver," Journal of Computer-Based Instruction V. 3, No 2, pp. 42-50.

Chapter 2

Basic Components of MUMPS

Character Set

The MUMPS language uses the seven-bit character code defined by the USA Standard Code for Information Interchange (ASCII) and published in the American National Standards document X3.4-1968. This character set, which is reproduced in Table 1, includes the 26 upper case letters A through Z, the 26 lower case letters a through z, the ten numeric digits 0 through 9, and the 33 special characters listed in Table 1. In addition, there are 33 non-printing control characters used in remote communications and other special actions. Programming may be done entirely in the 64 character subset of the ASCII code commonly found on teletypes, so that it is not necessary to have lower case alphabetic printing capability in order to use the language.

Numbers

MUMPS can be used to perform mathematical computations involving numbers. The simplest form a number may take is that of a positive integer. 17, 245, 1967, and 100500789 are examples of integers that are acceptable to MUMPS systems. Integers include whole numbers, i.e. those which have no fractional component. In MUMPS, an integer cannot be expressed with a decimal point; thus 13.0 is not an integer, even though it has a zero decimal fraction. Integers are defined as any whole number which may or may not be preceded by a positive or negative sign.

Most computing languages distinguish between an integer and a real number. A real number is defined as one in which there is a whole number and a decimal fraction, such as 2.3,7.5 or 100.500789. Real numbers are often used in calculations requiring greater accuracy, in division, and in other operations likely to generate a fractional result. MUMPS allows the user to distinguish between integers and real numbers in certain types of operations which will be described later. It is the user's responsibility to maintain this distinction in his programs if it will affect his results.

Table 1

USA Standard Code for Information Interchange (ASCII)
Character Set

Non-Printing Characters

Code (hex)	Character Name	Code (hex)	Character Name	Code (hex)	Character Name	Code (hex)	Character Name
00	NUL	08	BS	10	DLE	18	CAN
01	SOH	09	HT	11	DC1	19	EM
02	STX	0A	LF	12	DC2	1A	SUB
03	ETX	0B	VT	13	DC3	1B	ESC
04	EOT	0C	FF	14	DC4	1C	FS
05	ENQ	0D	CR	15	NAK	1D	GS
06	ACK	0E	SO	16	SYN	1E	RS
07	BEL	0F	SI	17	ETB	1F	US

Printing Characters

Code (hex)	Character Name	Code (hex)	Character Name	Code (hex)	Character Name
20	space	40	@	60	
21	!	41	A	61	a
22	"	42	B	62	b
23	#	43	C	63	c
24	$	44	D	64	d
25	%	45	E	65	e
26	&	46	F	66	f
27	'	47	G	67	g
28	(	48	H	68	h
29	)	49	I	69	i
2A	*	4A	J	6A	j
2B	+	4B	K	6B	k
2C	,	4C	L	6C	l
2D	-	4D	M	6D	m
2E	.	4E	N	6E	n
2F	/	4F	O	6F	o
30	0	50	P	70	p
31	1	51	Q	71	q
32	2	52	R	72	r
33	3	53	S	73	s
34	4	54	T	74	t
35	5	55	U	75	u
36	6	56	V	76	v
37	7	57	W	77	w
38	8	58	X	78	x
39	9	59	Y	79	y
3A	:	5A	Z	7A	z
3B	;	5B	[	7B	{
3C	<	5C	\	7C	|
3D	=	5D	]	7D	}
3E	>	5E	^	7E	~
3F	?	5F	_	7F	del char

Because most computers are binary machines, accurate representation of decimal fractions may differ slightly in the least significant components. Also, computers differ in the precise way that they handle fractions, so it is likely that dissimilar computers may produce results that differ slightly in the last digits printed. Although these minor discrepancies will probably not affect the significant arithmetic accuracy of many problems, it is worthwhile noting that such distinctions occur.

Humans find difficulty in reading long strings of digits. In conventional publications, large numbers are usually broken up by commas to facilitate reading, so that the number one million is often written 1,000,000. This type of notation Is not acceptable in MUMPS, nor is it allowed in most programing languages. Instead, a special notation used in scientific calculations has been adopted. This notation, called exponential notation, expresses the number raised to some power of ten by writing the number followed by E, then by the power of ten used. Here are some examples of exponential notation:

Conventional Notation	Exponential Notation
125	12.5E1 (12.5 times 10(1))
1389.468	13.89468E3 (times 10(2))
1000000	10.E5 or 10.0E5
172.4	17.24E1
0.0156	15.6E-3
.000000762	76.2E-8

Because this notation is useful in representing very large or very small numbers, it is used commonly for scientific computation.

In general, numbers represent a special case of character sets. In this PRIMER, we will refer to a set of characters satisfying the descriptions given above as a numlit, that is, a series of characters that can be interpreted as having numeric value. The following examples illustrate valid and invalid numlits:

Valid numexpr	Invalid numexpr
35	thirty-five
29000	29,000
10E5	10x10(5)
0.34	34/100
13.0E4	13.4 E4 (no spaces)
10	E+1 (no number)

Numeric Ranges and Precision

Digital representation of a fractional number is always finite; there can be no endless fraction such as the decimal representation of one-third. Computer languages must select some degree of accuracy for numbers used in computation. Standard MUMPS calls for nine digits of precision in any number, integer or real. Thus the numbers 123456789 and 1.23456789 are both valid, but a number such as 99.87654321 would probably be retained as 99.87654320, with the least significant number rounded off to zero. Since this Standard requirement is for a minimum of 9 significant digits accuracy, some Standard implementations may (and do) provide for even greater accuracy. If the problems you are dealing with require such precision, you should check with the provider of your system to determine the accuracy on your own machine (or, after you have progressed through a few more chapters, you might test the precision yourself).

Since exponential notation permits representation of much larger and smaller numbers than can be expressed by conventional notation, a convention must be adopted to set boundaries on the size of the exponent that must be recognized by Standard MUMPS. The ANSI Standard document specifies that numbers between 10E-25 and 10E25 will be accepted by any implementation of the Standard. Once again, many manufacturers offer ranges that exceed this minimum level. The precision of any number within the range will still remain at nine significant digits.

String Literals

Many uses of computers involve sorting and searching through natural language words - names, addresses, text materials, etc. MUMPS is particularly suited to the manipulation of such strings of characters. In general, a string literal is any combination of the printing ASCII character set illustrated in Table 1. Thus, all numbers defined in the previous section may also be considered as string literals (we will be calling a string literal strlit in cases where we wish to distinguish it from a numlit). The reverse is not the case, however, as shown earlier.

The convention in defining strlits is to enclose them in quotation marks. Thus "ABC" is a strlit whose three characters are A, B and C. Any string of numbers, alphabetic characters or special printing symbols can be included in a string literal. The convention of enclosing a strlit in quotation marks is fine for every case

except the one in which you wish to include a quotation mark in the string. However, the designers of the language, recognizing the validity of this option, modified the convention to define two quotation marks occurring together to mean a single quotation mark included in the strlit. When two quotation marks occur alone (i.e. not within a strlit), they signify a null (or empty) string. A few examples of the definition and actual strlits are given below.

Definition	Actual strlit
""	(null string)
"XYZ"	XYZ
"John Doe"	John Doe
"a ""super"" party"	a "super" party

Strings, like numbers, cannot be infinitely long, owing, among other things, to the storage capacity of the computer. Once again, therefore, limits to the length of strings have been placed in the ANSI Standard document. The upper limit has been established as 255 characters in any single string, and any Standard MUMPS implementation will handle strings of that length. Some manufacturers may have elected to extend these limits, but the need for longer strings is less immediate than for greater numeric precision, so most systems adhere to the Standard minimum.

Variables

It is common practice in algebra, for example, to substitute some letter (usually x in most textbooks) for an unknown quantity in an equation. Thus, we may be asked to state the value of x in the equation x = 13+22. This approach is valuable in many problems where a variable will take on different values under different circumstances. Computer languages provide various ways of defining such variables. In MUMPS, a variable may be named anything, provided that the first letter is a capital letter of the alphabet or the character "%" (percent sign), and the remaining characters are either capital letters or numbers. Acceptable names might include X, NAME, %HGHT, or R2D2. Names such as z,25th, or A*B are not valid according to the definition provided above. Even though it might be possible to use lower case letters in names, the practice is discouraged because many terminals cannot accept or enter the lower case alphabet.

Although a variable name may be arbitrarily long, it is not

generally useful to assign an extremely long name to a variable that is often used in manipulating data. Standard MUMPS specifies that a name can be up to 8 characters long; any letters in a name beyond the first 8 characters are ignored. Once again, some implementations may allow for longer names, so you should check your system if you are interested in creating longer names than provided by the Standard.

Many languages (such as FORTRAN) adopt special rules distinguishing integer variables, real variables, and other types. MUMPS does not make any such type distinction in storing variables. This is an important feature distinguishing MUMPS from many other programming languages. It means that numbers inside a strlit may, using some of the language features described later, be isolated, used in calculations and then replaced in the string for subsequent storage or output. The practical applications of this feature will be illustrated in later chapters of this PRIMER.

Functions

Up to now, we have described the basic building blocks of the MUMPS language - numbers, strings and variables. The power of many languages, including MUMPS, is often dependent on special operations that assist in the manipulation of these basic elements. One class of such operations are referred to as Functions, and they may be recognized by their names, which always begin with a dollar sign ($). Some of the functions that will be described in detail in later sections of this text are $FIND, $PIECE and $SELECT. For the present, if is sufficient to understand that such functions exist and that they may be used in performing a variety of operations on string and numbers in the MUMPS language.

Special Variables

The last component that we will consider in this chapter describing the MUMPS building blocks is that class of elements called Special Variables. Like Functions, these elements are identified by a name beginning with a Dollar sign. Unlike Functions, however, these elements simply provide a single value, such as the date and time ($HOROLOG), the positin of the cursor on a line of text ($X), etc. Special Variables will be more completely introduced in later chapters.

Summary

In this chapter we have defined the basic elements of the MUMPS

language - numbers, strings, variables and functions. Variables may include both numeric and character components. Later, we will learn about special Functions and other language elements that make it possible to manipulate individual components of such variables in order to solve a wide variety of complex problems.

Chapter 3

Operations on Numeric Values

In Chapter 2 we learned that there are two basic types of data defined in the MUMPS language: numbers and strings. You also learned that both types can be stored in the same variable at different times. This chapter introduces the types of operations that can be performed on numeric values; later, we will learn about other operations affecting strings as well as numbers. Most of the concepts used in MUMPS should already be familiar, since these ideas are really ones that you learned in elementary arithmetic.

Binary Operators

MUMPS defines six operators which are used to combine numeric values in different ways. They are:

+	addition
-	subtraction
*	multiplication
/	division
\	integer quotient division
#	modulo, or remainder division

The first four operators correspond to the four basic arithmetic operations used in everyday activities. In their most common form, these operators are binary in nature; that is, they act on two values to produce a third. Thus

```
4+7     =  11
18-6    =  12
3-4     =  -1
9*6     =  54
10/2    =  5
4/3     =  1.33333333
```

The last example is included to reinforce the point that MUMPS preserves nine significant digits. The quotient of 4/3 is a number of infinite length, 1.3333...; however, MUMPS, as noted earlier, truncates this value to nine significant digits.

The remaining two operators have to do with different forms of division. Integer quotient division, as indicated by its name, returns only an integer value. Strictly speaking, the operator \ returns only the integer portion of the quotient calculated by the more normal (/) division operator. A few examples should make the point more clearly (ignoring for the moment the last column in the table):

N	D	N/D	N\D	N#D
10	2	5	5	0
-8	4	-2	-2	0
6	-3	-2	-2	0
-6	4	1.5	1	2
1	3	0.33333333	0	1
-14	-6	2.33333333	2	-2
4	-3	-1.33333333	-1	-2
13	5	2.6	2	3

Modulo,or remainder division is the complementary operation to integer division, in that the value returned by the # operator is the remainder deleted by theinteger division. For positive numbers, the result of modulo operation is the familiar remainder from long division. Thus 13 divided by 5 gives a quotient of 2 and a remainder of 2 (the modulo results in the last line of the table given above). A formal mathematical expression of the modulo relationship is given by the equation

$$N\#D = N - (D * \text{floor}(N/D))$$

where N is the numerator, D the denominator and floor(X) is defined as the largest integer less than or equal to X. Substituting the values from the last two examples given above, we obtain

```
4-(-3*floor(4/-3))
4-(-3*floor(-1.33333333))
4-(-3*-2)
4-(6)
= -2
```

and, in the next example,

```
13-(5*floor(13/5))
13-(5*floor(2.6)
13-(5*2)
13-10
= 3
13-((13\5)*5)
13-(2*5)
13-10
=3
```

Using this definition, you should be able to verify each of the relationships in the examples given above, remembering that the sign of the result is always the sign of the denominator.

Arithmetic Expressions

The arithmetic operators defined above may be combined in various ways to form arithmetic expressions, statements involving arithmetic values and arithmetic operators that may be evaluated to give a single numeric value. The previous section gave examples of several simple expressions, each consisting of a single operator between two values. Expressions may, however, be considerably more complex. For example,

```
1+2+3+4+5+6=21
```

is an expression that uses several values and several operators, yet still gives a single result. As expressions become more complex, involving different operators, computer languages like MUMPS must define rules for evaluating them in order to give consistent results. Consider the expression 3+5*2. As it is written, there are two possible interpretations of the expression, depending on which operator is evaluated first. The alternatives are clarified by rewriting the expression with parentheses, so that the meaning may be either (3+5)*2 to give a result of 16, or it may be 3+(5*2), which produces a result of 13. In MUMPS, as in other computer languages, you may use parentheses, controlling the sequence of operations to produce the desired result. However, this approach fails to tell us what happens if no parentheses are used in such expressions. MUMPS simply evaluates such expressions from left to right, taking each value and operator in sequence as it is encountered. (Note that this left-to-right precedence is not used in some other computer languages). Thus, our example is treated by the MUMPS language as if

it had been written (3+5)*2, or 16. Naturally, parenthesized expressions are evaluated first, so if you wanted the alternative interpretation, you would have to use parentheses to obtain it, i.e., 3+(5*2). If MUMPS saw this equation, it would evaluate the parenthesized value first, giving the final result of 13.

Left-to-right evaluation appears natural and causes no problems to programmers unfamiliar with other computer languages. The experienced programmer in a language such as FORTRAN, however, will immediately notice that the precedence defined by MUMPS is different from FORTRAN, which assigns higher precedence to multiplication and division than to addition and subtraction. This difference can always be resolved by using parentheses, so you should use them freely if you have any doubts about a particular calculation.

Let us conclude this section with one final example illustrating the order of execution of a complex statement. In each line, the underlined operation is the next one performed.

17-4/2+(8*3)\6#2

13/2+(8*3)\6#2

6.5+(8*3)\6#2

6.5 + 24 \6#2

30.5 \ 6 #2

5#2

1 (final answer)

In all of the previous examples, we have used numeric constants to keep the examples as simple as possible. MUMPS also allows us to use variables in arithmetic expressions. If, for example, the following assignments had been made to a set of variables:

A=17 B=4 C=2 Y1=3 BCX=6 X=8

then the previous example could be rewritten

```
A-B/C+(X*Y1)\BCX#C
```

and it would be evaluated in precisely the same sequence as shown earlier, producing the same result. Any arithmetic expression may be composed of a mixture of arithmetic constants, arithmetic variables and arithmetic operators together with parentheses required (or desired) to clarify the evaluation.

Unary Operators

Two of the arithmetic operators (+ and -) may also appear in front of a single operand. In such a case, instead of being binary operators requiring two operands, they are referred to as unary operators. These unary operators are used especially in dealing with arithmetic variables, converting them from string to numeric form when there is any question about their string representation. The unary plus sign simply converts a string to its numeric equivalent; the minus sign changes the sign of such a conversion. If, for example, we have three variables A=0, B=3 and C=-6, the result of unary operations on these variables would be as follows:

```
+A  =   0
-A  =   0
+B  =   3
-B  =   -3
+C  =   -6
-C  =   6
```

Numeric Interpretation

With one exception, these examples complete our discussion of arithmetic expressions. Remember, however, that a variable may contain either a numeric value or a string value. In an expression such as A+3, the definition of addition requires that A have a numeric value. In general, you need not worry about that fact, since MUMPS incorporates a feature known as automatic mode interpretation, which means that if A contains a string value when the expression A+3 is evaluated, MUMPS will convert the string for you before evaluating the expression. Thus, if A has the string value "123", the result of A+3 would be 126. You need not remember the actual conversion rules

used by MUMPS in converting a string to a number; they are spelled out in the language Standard for reference. Instead, we will provide a simpler rule, based on the earlier definitions of numbers. MUMPS considers the string from left-to-right. If only the left-hand portion of the string (after removal of any leading unary operators) forms a valid numeric constant, the string is interpreted as being tht numeric value. If no valid numeric constant can be found, MUMPS assigns the value of zero to the numeric interpretation. A few examples will perhaps clarify these rules:

String	Value
"ABC"	0
""(null string)	0
"123"	123
"-12"	-12
"+36.5"	36.5
"12ABC"	12
"3.1E-3"	.00031
"A12B36C"	0
"+-0.362"	-0.362
"9E2AB"	900

One of the most common uses of the unary operators is to force a numeric interpretation of its operand. There are situations in MUMPS where a numeric value is absolutely required. If these situations, use of a unary plus will force MUMPS to perform a numeric interpretation. In the examples given above, addition of the unary + in front of the each of those strings (or their variable names, if any) would produce the corresponding value shown in the table.

Arithmetic Functions - $RANDOM

Standard MUMPS defines only one purely arithmetic function: $RANDOM. This function takes a single argument, an integer that must be equal to or greater than one. Each time the function is called, it returns a random integer within the range 0 to (argument -1).

Thus, calling $RANDOM(100) returns a value between 0 and 99, $RANDOM(10) an integer between 0 and 9, and $RANDOM(1) returns only the value 0 (and hence is pointless). This function is useful in a number of arithmetic calculations where random values are required.

Summary

In this chapter we have introduced the concept of relational operators, selecting arithmetic operators as the simplest examples. We move next to consideration of two other families of operators, those that are logical or comparative, and those that involve string manipulations. These concepts will be covered in the following two chapters.

Chapter 4

Logical Operators, Comparisons

In the previous chapter, we considered arithmetic expressions that allow us to set the value of a variable or to perform calculations using variables and constants. In many computing applications, however, we need to test a relationship between two variables and take differing actions depending on the result of the test. All computer languages have some way to perform tests of this type. In MUMPS, there are several different types of comparisons and operations that can assist in this process. These operators will be considered in this chapter.

Arithmetic and Relational Operators

Since MUMPS stores variables as strings, a distinction must be made between the value of numeric strings and strings themselves, since some numbers may be expressed in several different ways. The numbers 35, 3.5E1 and .35E2, for example, are all equivalent, but they are quite different in the way in which they are stored. To test numeric relationships, two operators are available:

<	less than
>	greater than

When these operators are invoked, the variables are first converted to their numeric value, then the comparisons are made.

In addition to the two arithmetic relational operators, three additional string operators are used. They are

=	equals
[	contains
]	follows

The first of these tests to see if two strings are equal. Note that this comparison is not numeric, so the numbers 35 and 3.5E1 are not evaluated as equal using this operator. If numeric equality is to be tested, the programmer may make use of the unary + operator to force numeric interpretation. Thus, if

A = "35" B = "3.5E1"

then

A=B is not true, but

+A=+B

is true, since numeric interpretation is forced. Programmers should take care in using this operator when numeric values are involved.

Note that when numeric values are produced as the result of previous numeric operations, e.g. addition, the resulting strings are in MUMPS "canonic" form. Since there is only one such form for each numeric "value", the equals operator may be applied directly. If A="33"+"2" and B=+"3.5E1" then

A=B

is true.

The contains operator will test to see if a value contains another value. If VAR is assigned the value "ABC123" and TEST1 and TEST2 are assigned values of "BC1" and "C2", then some possible tests and results would be as follows:

VAR["A"	evaluates as true (A is in VAR)
VAR[TEST1	evaluates as true (VAR contains BC1)
VAR[TEST2	evaluates as false (although "C" and "2" are both in VAR, they do not occur together)

When we say that one variable follows another, we are testing its collating order, using the ASCII collating sequence illustrated in Table 1. Using the three variables above,

VAR]TEST1	evaluates as false
TEST1]TEST2	evaluates as false, and
TEST2]VAR	evaluates as true

This operator is particularly useful when it is desired to sort names or other strings into an alphabetic sequence, since the letters of the alphabet are properly ordered in the ASCII collating sequence. Comparison is done on a character by character basis, from left to right.

When we say that a test of this sort evaluates as true or false, we imply that the result of a test such as those given above will return the value 1 if the test is true, and 0 otherwise. These values, 0 and 1 are referred to as truth values.

Logical Operators

In addition to the arithmetic and relational operators described in this and the preceding chapter, MUMPS allows for testing combinations of conditions using the logical operators

&	and
!	or
'	not

The result of an and test will be true only if both tests are found to be true. An or test will return a true value if either condition is true. Given a situation where X has the value "ABC" and Y the value "DEF", we can demonstrate the combination of logical operators with some of the earlier expression tests as follows:

"ABC"=X	True (1)
"AB"=X	False (0)
X["AB"	True
"AB"[X	False
X[Y!(X["AB")	True
Y]X&(Y["EF")	True
X]Y&(X["B")	False
12>11!(X=Y)	True

The not operator (') is used to reverse the truth value of the affected expression. Thus '(11>12) would evaluate true, whereas '("ABC"="ABC") would evaluate false (the reverse of its original truth value). The not operator may also be used in conjunction with the relational and logical operators. For example,

11'>12	is true, and
"ABC"'="ABC"	is false.

In general A 'relation B has the same value as '(A relation B).

Using the operators described above, complex combinations of relationships may be tested, and a single true or false value returned.

Summary

This chapter has introduced the MUMPS relational operators. Together with the arithmetic operators described in the previous chapter, they represent a powerful set of tools that permit the programmer to perform many tasks involving numeric and logical relationships.

In the next chapter, we will introduce a final set of operators, those that allow MUMPS users to manipulate strings. These three

chapters together complete the principal manipulative tools of the MUMPS language. Since MUMPS is designed particularly with the intent of permitting string manipulations, it is the final chapter in this series that contains those features that distinguish MUMPS from many other programming languages.

Chapter 5

Operations on Strings

Introduction

In the previous two chapters, we considered first, arithmetic operators and functions, then logical and relational operators. This chapter is concerned with the final group of operators and functions, those used for string manipulation. These tools complete the library available for the MUMPS programmer. In this chapter, we consider all string operations except for Pattern Match, which is discussed separately in Chapter 13.

Strings or text may be manipulated in a number of ways. Substrings may be located, isolated, or moved to other strings. A string may be appended to another string. Strings may be tested for their length, the occurrence of specific characters, or compared, using the relational operators described in the previous chapter. Since MUMPS was designed to allow extensive text manipulation, all these functions and more can be readily accomplished by using the functions and operators described in this chapter. These concepts can be understood thoroughly only through application; the reader is urged, therefore, to practice the capabilities describe below with problems of his own devising.

String Operators

In Chapter 4 we presented three operators which act on strings. They were

Operator	Action taken
A=B	compares A with B for equality
A[B	determines if A contains B
A]B	determines if A follows B in the collation sequence

To this group of operators, we now add a fourth, represented by the symbol _ (underline). The purpose of this function is to append one string to the end of another, or to concatenate the strings. The following examples describe its use with literals and variables. Assume that A="ABC", B="MUMPS".

Expression Evaluated	Value Produced
A_"DEF"	ABCDEF
A_B	ABCMUMPS
A_"'S OF "_B	ABC'S OF MUMPS

This command is particularly powerful in constructing messages into which string variables are inserted. For instance, if a variable called USER had been set equal to the name of the user at the terminal, we might create the following string:

"HELLO, "_USER_". WELCOME TO MUMPS."

the resulting string would concatenate the three strings together, inserting the user's name in the middle.

String Functions

In Chapter 2 we considered the $RANDOM function and its use in arithmetic operations. We now examine several functions that provide the tools necessary for effective string manipulation.

$LENGTH Function

One important characteristic of a string is its length, i.e. the number of characters it contains at a given time. $LENGTH determines the length of any string, returning a single number which may range from 0 to 255, the maximum string length allowed in MUMPS. Here are some examples and their actions, assuming that T="JOE BROWN", V="ABC;DEF;GHI", DAT="9/5/55", and X="TOWN":

Expression Evaluated	Value Produced
$LENGTH("") (null string)	0
$LENGTH(T)	9
$LENGTH(V)	11
$LENGTH(DAT)	6
$LENGTH(T_X)	13
$LENGTH("THE END")	7

Note that ($LENGTH(T)+$LENGTH(X)) is the same as $LENGTH(T_X). In other words, an expression, rather than a single string or variable, may be evaluated by the $LENGTH Function.

$FIND Function

The $FIND function determines whether and at what position a substring occurs within another string. It is often used when the location of data within a string is unknown. Some distinctive portion or even the entire field may be used as the second argument of $FIND which, upon evaluation, indicates the position of the second argument within the string specified in the first argument. The result is an integer value which corresponds to the character position immediately following the occurrence of the second argument string within the first. For all instances where the second argument substring does not appear within the first argument string, a zero value is returned by $FIND.

Consider the following, where variables have the values S="ABABCDEFABCXYZ" and SS="ABC"

Expression Evaluated	Value Produced
$LENGTH(S)	14
$FIND(S,"AB")	3
$FIND(S,SS)	6
$FIND(S,"PQ")	0

$FIND may also have a third argument which specifies the character position within the first argument string at which the search for the second argument substring is to begin. Thus, using the same variables defined above:

Expression Evaluated	Value Produced
$FIND(S,"AB",2)	5
$FIND(S,"AB",4)	11
$FIND(S,SS,4)	12
$FIND(S,SS,10)	0

$EXTRACT Function

The $EXTRACT Function is employed primarily for the extraction of fixed length subfields from a composite string. Like $FIND, $EXTRACT can have either two or three arguments. A two-argument form of $EXTRACT is used to extract a single character from the first argument, at the character position specified by the second argument. If, for example, S has the value "ABCDEFGHIJ",

Expression Evaluated	Value Produced
$EXTRACT(S,5)	E
$EXTRACT(S,7)	G
$Extract(S,$L(S))	J
$EXTRACT(S,0)	null
$EXTRACT(S,11)	null

In the three-argument form of $EXTRACT, the second and third arguments specify the initial and final character positions, respectively, of the desired field which is to be extracted. Continuing with our previous example,

Expression Evaluated	Value Produced
$EXTRACT(S,2,5)	BCDE
$EXTRACT(S,6,$L(S))	FGHIJ
$EXTRACT(S,7,7)	G
$EXTRACT(S,8,7)	null

In the last example, since the third argument is less than the second, no string is specified and the result is null.

$PIECE Function

The $PIECE function provides an increased facility for separating subfields of a string which contains one or more delimiting characters. In contrast to $EXTRACT, which requires knowledge of the initial and final positions of the field to be extracted, $PIECE will search a string for a given delimiter, regardless of its position in the string.

With $PIECE, all that must be known in order to retrieve a substring field is the relative placement of that field with respect to other fields, and the delimiter used to separate these fields in the main string. The first argument of $PIECE is the string from which the subfield is to be retrieved. The second argument is the delimiter, and the third is the relative field position of the desired field. To illustrate, let us define SD="LASTNAME,FIRSTNAME" and S="ABC/DEF/GHI". By using two different delimiters for the two variables selected,

Expression Evaluated	Value Produced
$PIECE(SD,",",2)	FIRSTNAME
$PIECE (S,"/",3)	GHI
$PIECE(S,"/",1)	ABC
$PIECE(S,"/",4)	null

Like $EXTRACT, another argument can be added to $PIECE to signify a range of fields to be extracted. The following examples

illustrate this form of $PIECE (using the same strings defined above):

Expression Evaluated	Value Produced
$PIECE(S,"/",1,2)	ABC/DEF
$PIECE(SD,",",2,5)	FIRSTNAME

The value of the delimiter is not limited to single characters of punctuation marks. Consider the following examples where X="A":

Expression Evaluated	Value Produced
$PIECE(S,"DE",1)	ABC/
$PIECE(S,X,2,3)	STNAME,FIRSTN

It is relatively easy, using this function, to retrieve subfields from larger data strings, thereby permitting the MUMPS programmer to store concatenated variables for subsequent decomposition. $PIECE is, in fact, one of the most useful string Functions in the MUMPS language, performing a function that requires complicated special coding in most other computer languages. Examples of its use will appear in later chapters of this PRIMER.

$ASCII Function

Sometimes it is valuable to know the ASCII code representation of a particular character in a string. $ASCII provides this value, using two arguments, the first giving the string to be searched and the second giving the position of the character to be evaluated. If no second argument is used, the first character in the string is evaluated. The number returned by $ASCII will be some value between 0 and 127 (base 10), the range of ASCII character codes. If we have two variables assigned values XX="ABC123" and YY="Q:R!", then

Expression Evaluated	Value Produced
$ASCII("A")	65
$ASCII(XX)	65
$ASCII(XX,4)	49

$ASCII(YY,2)	58
$ASCII(YY,4)	33
$ASCII("")	-1

Note that, since all values between 0 and 127 are valid ASCII characters, the number -1 is used to represent the null string.

$CHAR Function

The $CHAR Function performs the inverse of $ASCII; it returns a character at the collating sequence position specified by its argument value. Argument values must be integers less than 128 in order to avoid an error. Negative argument values are ignored. For multiple arguments, the characters which correspond to each argument are returned in the same order, as a single string:

Expression Evaluated	Value produced
$CHAR(72,73,89,65)	HIYA
$CHAR(77,65,-40,67,75)	MACK
$CHAR(-10)	0

Summary

In this chapter we have covered the operators and functions commonly used in manipulating strings. Examples illustrating their use in practical applications will be given in subsequent chapters, when we have learned how to create complete statements in MUMPS.

Chapter 6

Introduction to Commands

What is a Command?

Consider the following three English sentences:

Stop.

Open the door.

Put the book on the table.

Each sentence is a command, telling someone to do something. In speaking to other human beings, we generally soften these kinds of orders by saying "Please open the door," but when we do so, we introduce some shades of meaning that make the request a little more ambiguous. "Please stop" may mean something quite different under certain circumstances than "Stop!"

When we tell a computer to do something, we want to be as clear and unambiguous as possible, so we use direct (hence impolite) terms to express our desires. Fortunately, the computer prefers this type of communication, since it might be confused by any other form.

The English sentences listed above are all unambiguous commands, if we assume that only one door, book or table are available. The first sentence is only one word long. It is simply a command to do something. In the second case, we are telling someone to do something with something. Finally, in the third sentence, we tell someone to do something with something and something else. We could devise considerably more complex commands, but these examples will suffice for the moment.

A computer command is like a direct English command in many respects. Some Commands are one word long, such as HALT (we'll describe this command in a later chapter). Other Commands require the computer to perform some action on one or more elements available

at the time. In these latter cases, we say that the Command has arguments. A command may have no arguments, one argument, or more than one argument. When multiple areguments are present, they are separated by commas.

SET Command

Let us take the MUMPS Command SET as an example. SET may be used to assign a value to a variable. SET X=3 assigns the value of 3 to the variable X. If the variable X did not previously exist, it is created and the value 3 then assigned to it. If the variable did exist previously, having, for example, the value "ABC", execution of the SET command causes the previous value (ABC) to be lost and the new value (3) to replace it. Since MUMPS does not distinguish different types of data, e.g. numeric or string, any value may be assigned to any variable at any time. SET NAME="JOHN DOE" assigns the strlit JOHN DOE to the variable NAME. If we make use of arithmetic operators described in the chapter 2, we could, for example,

```
SET Y=X+7
```

If X was equal to 3 when this command was executed, then Y would be assigned the value 10. If X is undefined, an error message would result.

MUMPS allows us to use the SET Command in several ways. We can have a simple SET, such as the examples given above. We can also assign new meanings to several variables at once, using a multiple SET. For example,

```
SET (X,Y,Z)=20
```

would assign the value 20 to each of the variables X,Y and Z. Similarly,

```
SET X=5,Y=2,Z=X*Y,W=X>Y
```

In this case, whether or not X had some previous value, it would be reassigned the value 5, and Y the value 2. Z would then be set

equal to their product, or 10. W would take on the truth value one, since the expression X>Y is true. This example demonstrates that when commands are given multiple arguments, the arguments are executed from left to right, one at a time.

KILL Command

Once a variable has been defined using the SET Command, it may be removed from the list of variable names and their values by using the KILL Command. Like SET, KILL may be used in several ways. When used with a single argument, KILL removes that variable and its value from the list of values available to the programmer. KILL followed by two or more arguments removes each of the variables specified. Finally, KILL used without any arguments removes all of the variables available at the time. Thus,

Code Executed	Effect Produced
KILL A	variable A and its value are deleted
KILL B,C	variables B and C are deleted
KILL	all variables are deleted

Note that in the uses of KILL specified above, no parentheses are used to identify the variables to be deleted. Another form of KILL exists, in which parentheses are used, to indicate which variables are to be preserved after the KILL is executed. This is equivalent to saying "KILL all except (...)." Examples are:

Code Executed	Effect Produced
KILL (A)	all variables except A are deleted
KILL (B,C)	all variables except B and C are deleted

Note that the last example is not a multiple argument form (outside parentheses) and it is therefore not the same as KILL(B),(C).

$SELECT Function

To consider another example, let us examine the special function $SELECT. The $SELECT function selects, from a list of expressions, a single expression that is evaluated and whose value is returned. To illustrate, consider the following situation, where variables have been set as follows: A=1, B=2 and C=3. If we want to establsh the value of a new variable, RESULT, based on the $SELECT function, we might use the following example:

```
SET RESULT=$SELECT(A>B:1,A=B:2,A<B:3)
```

The MUMPS system would invoke the $SELECT function and examine each alternative presented between the parentheses, moving from left to right until a true expression is found. Since, in our example, A is less than B, the first two cases would be ignored and the third case selected. Since the third case is true, the variable RESULT will be assigned the value 3 as specified.

$SELECT can also be used with a final case which is a default option, selected only when all other cases do not fit the situation at hand. The syntax of this last case is shown in this example:

```
SET RESULT =$SELECT (A=B:4,A=C:3,1:0)
```

In this case, since A is neither equal to B nor to C, the default choice of 0 is assigned to the variable RESULT. The $SELECT function must always have at least one true case or an error will result. When a situation arises where the computer cannot do what it has been ordered to, the MUMPS system will indicate that an error has occurred. It always assumes (in its superior way) that the error is the fault of the programmer, and tries to help him find it with some cryptic message. Error messages vary from one system to another, but it is nice to know that the computer is looking out for us, and that it will point out those errors that prevent it from doing what we ask.

Let us examine a few additional examples of use of the $SELECT function. Again, let A=1, B=2, and C=3.

Code Executed	Value Produced
SET RESULT=$SELECT(A>B:1,A>C:2,A<B:3)	3
SET RESULT=$SELECT(A=B:1,B=C:2,A=C:3)	error
SET RESULT=$SELECT(A=B:2*C,A=C:A+B,1:0)	0
SET RESULT=$SELECT(A>B:A,A<B:C,A<C:B)	3 (value of C)
SET RESULT=$SELECT(A=C:"LB1",A<C:"LB2",1:"LB3")	LB2

Note that in the fourth example, two true cases are present. $SELECT will choose the first true one as it proceeds from left to right.

WRITE Command

Thus far, we have examined a typical Command and a Function. Let us now look at another type of Command and some of its options. The example we will use is WRITE, which can be used to direct messages to the user's terminal. In its simplest form, WRITE can merely repeat a string: WRITE "George Washington crossed the Potomac" will reproduce that sentence on the output terminal. If we want to print the result of some calculation we could use the form

```
WRITE "THE VALUE PRODUCED IS : ",X
```

where X has been calculated earlier. The resulting output would contain both the message and the value currently assigned to the variable X. This example shows why it is necessary to enclose strings in quotation marks, since otherwise the computer would have difficulty in distinguishing between strings and variable names.

In order to allow the user to specify where to place the WRITE message on the terminal's screen or page, we can make use of several Format Controls. The Control Character "!" appearing by itself (without quotation marks) in a WRITE statement will cause the remainder of the message to be printed at the start of the next line. Thus, WRITE "HELLO",!,"MY NAME IS ",NAME will produce two lines of output, the second beginning with MY NAME IS and then concluding with the value assigned to the variable NAME. Note that the first column position in a line is position 0, and likewise, the first line on a

page is also 0.

Similarly, one can use the special character "#" to move to a new page on printed output or the top of the screen on a CRT terminal. The special character "?", followed by an integer, will position the message starting at the column specified by the integer. The example

```
WRITE #!!,"CLIENT",?10,NAME
```

executed on a CRT terminal would clear the screen, move the cursor to the top left position on the screen (initiated by #), drop down two lines (initiated by !!), write the word CLIENT, then skip to column 10 and enter the current value of the variable NAME.

$X and $Y Special Variables used with WRITE Command

Sometimes the user will not know beforehand exactly how long some variable such as NAME may be at output time. In this case, it would be nice, for cosmetically acceptable output, to calculate the position of the printing element so as to adjust accordingly. Two Special Variables, $X and $Y, are provided by MUMPS to give the user this information. $X indicates the position of the cursor or printing element on the current line of text, and $Y indicates the number of lines down from the top of the screen or page. With these two special variables available to us, it is possible to make a neat output line even when variables take on different lengths.

READ Command

The WRITE Command has been introduced before the READ Command because, in effect, READ allows the user to write a message to the terminal at the same time that it requests input from the user. This capability is particularly useful to avoid those long awkward pauses, where the computer is awaiting input from the user, but no clue has been provided that this is the case. In this "writing" aspect, READ commands are constructed just like the WRITE Command described above. What is added, however, is the name of the variable or variables that are to be entered by the user at the terminal. The following example illustrates a READ Command with appropriate prompts:

```
READ !,"AGE= ",AG,!,"DATE OF BIRTH (MO/DA/YR)= ",DOB
```

With the execution of the READ Command, any string literals included in its argument list will be displayed at the user's terminal. The output format characters #,!, and ?expr perform as in the WRITE Command. In our example, the printer would advance one line, print the first message, wait for input from the user, then advance another line and print the second message before awaiting input for the variable DOB. Specifically, READ accepts input from a device and assigns this input to a specified variable. The value assigned is a sequence of characters up to, but usually not including, an implementation-specific terminator. Execution of further commands is suspended until the terminating condition has been met.

Postconditionals on Commands

MUMPS provides the programmer with the capability of conditionally executing a Command, based on some value or expression which may be evaluated as a truth value - either true or false. This operation is called a postconditional. Take, for instance, the Command

```
SET X=X+1
```

Each time the command is executed it will add one to the current value of X, then replace X with the new value. Suppose, however, that you only wanted X to reach a maximum value of 10, after which no further incrementing of X is to be performed. One way to accomplish this objective is to add a postconditional expression to the original Command. The new Command would then read

```
SET:X<10 X=X+1
```

The added characters :X<10 are the postconditional, where the colon (:) indicates the presence of a postconditional expression, and X<10 is the expression itself. When the command is executed, MUMPS detects the presence of the postconditional and suspends execution of the SET Command temporarily until it has evaluated the postconditional expression. If the value is true, MUMPS continues with execution of the SET Command. In our example, should the boolean value of the postconditional expression be false, MUMPS will ignore the SET Command, not increment X, and X will thus remain at 10 when the next Command in sequence is executed.

The postconditional expression may be more complex than the example given above. It may, in fact be any legitimate MUMPS expression. Postconditionals may be applied to all but three MUMPS Commands (IF, ELSE and FOR, which will be discussed in a later chapter). Combined with the WRITE Command, postconditionals give flexibility in output depending on the circumstances specified. As one example, suppose you wish to process a large data set, monitoring progress as you go along. One way to monitor progress would be to write an output statement each time a data element had been processed. This approach would probably tell you more than you wanted to know, generating voluminous output on the terminal screen or printer. A better way might be to have the program write a message each time it had processed, say, 100 items. The output would be reduced a hundredfold, giving you adequate information about the processing without excessive output. If the number of items processed is defined by incrementing a variable N, the following command, inserted after N has been incremented, would generate the desired output every hundred iterations:

```
WRITE:N#100=0 !,N," ITEMS PROCESSED."
```

Note that the postconditional applies to all arguments of the command. Although the postconditional is useful in a number of other situations, we will defer discussion of them until the appropriate Commands have been introduced.

Abbreviated Forms of Commands

MUMPS allows two ways of writing each Command or Function. The first is to write the entire word, such as SET, KILL, WRITE, $SELECT, etc. The second method is to write only the first letter of the Command, or the first two letters of Functions. The Commands and Functions we have considered thus far may therefore be abbreviated as follows:

KILL	K
READ	R
SET	S
WRITE	W
$ASCII	$A
$CHAR	$C

$EXTRACT	$E
$FIND	$F
$LENGTH	$L
$PIECE	$P
$RANDOM	$R
$SELECT	$S

Examples of the use of abbreviated Commands are:

```
R "PLEASE ENTER TODAY'S DATE",DAT
W !!,"TODAY'S DATE IS ",?20,DAT
S A=3,B=A#2,C=A<1,R="XYZ"
```

Note that, in their abbreviated form, single letters may look like single letter variables. The R in the last line, for example, is a variable, not a Command. MUMPS can make these distinctions by the position of the abbreviated Command in context. In this PRIMER, however, we will stick to the full spelling of the Command so as to minimize the opportunity for confusion.

Another characteristic of MUMPS which is in part the result of the abbreviation capability expressed above, is that spaces are often important in their presence or absence in a Command. If you will review the examples in this PRIMER, you will see that spaces are not allowed, for example, between multiple SET expressions, but that a single space is required between a Command and the remainder of its statement. In certain cases, discussed later, two spaces are necessary to preserve the syntax of a statement. Programmers familiar with other languages where blanks may be inserted with relative freedom will want to pay especial attention to this distinction.

Summary

Having considered arithmetic, relational and string operators in previous chapters, we next introduced the concept of Commands,

selecting several Commands and Functions to illustrate the general principles. In the next chapter we will see how these Commands may be combined into groups of instructions defined as Command Lines.

Chapter 7

Command Lines

Format of a Command Line

We have learned, so far, how to take the "Nouns" and "Verbs" of the MUMPS language and to construct statements with the vocabulary already introduced. Next, we will turn our attention to creation of groups of statements (Commands) which link together to perform useful functions.

The basic unit of an executable MUMPS program is a Command Line. This unit, in turn, is made up of the elements introduced in previous chapters - Commands, syntactically linking variables, operators and functions. In addition, a Command line may contain other elements (labels and comments) for procedural or clarification purposes. A Command Line may consist of a single Command; the examples in the previous chapter were valid Command Lines. However, a Command Line may also combine multiple commands. The length of a Command Line is limited, not by the physical length of a line of text on a terminal, but by the MUMPS limitation of 255 characters to a string (since Command Lines are stored as strings).

A Command Line may contain three different components : a label, one or more commands and a comment. Any or all of these elements may be missing in a Command Line. A line which contains no commands may be used for procedural or documentation pusposes, or even to improve readability of the code. Command lines may contain up to 255 characters. They are terminated by an end of line code, consisting (in most implementations) of the ASCII Carriage Return (CR) and Line Feed (LF).

A label is a name appended to the beginning of a line for reference purposes. Labels may either follow the rules for names of variables (i.e., begin with % sign or upper case alpha letter), or they may be integers. Real numbers are not permitted as labels. Since a label must uniquely identify a specific Command Line, provision must be made to ensure that no two labels are the same within a given block of MUMPS code (we will define such a block in Chapter 8). A Command Line may have only one label (and that one is optional); the

label, if present, must occur at the beginning of the line.

The only required part of a Command Line is the line start character. This character (a space in many implementations, a tab in some others) separates the label field from the rest of the command line. Commands and comments, if they are present, follow the line start character.

A comment consists of a semicolon (;) followed by any free text the programmer wishes to add to help the reader understand the purpose of the Command Line. This feature permits internal documentation of MUMPS code so that its purpose and relationship to other blocks of instruction are clarified. When the MUMPS interpreter encounters a semicolon, it will automatically ignore the remainder of that line. Therefore, a comment must always be at the end of any Command Line on which it appears. Sometimes, programmers will write lines which consist only of comments (i.e., the first character of the line is a semicolon).

To illustrate the concept of a Command Line, let us consider a few examples. First, let us examine the following line

```
START READ "ENTER WEIGHT (LBS)",L SET K=L/2.2 WRITE K," KILOS, RIGHT?"
```

This Command Line contains a label and three separate commands. The first command requests the user to enter his weight in pounds. It then calculates the equivalent weight in kilograms, and finally prints out a message to verify the accuracy of that calculation. Notice that the line begins with a label followed by a single space and that single spaces separate the individual commands (commas have a different meaning in MUMPS syntax and cannot be used for this purpose). As presented, this Command Line is a complete miniature program, with input, calculation and output. Remember that a "Line" may extend over more than one physical lines of text, provided that the 255 character limit is not exceeded.

Another major point to bear in mind with respect to Command Lines is their left-to-right execution. The example given above would make no sense if this sequence were not followed, and, as one might expect, all Command Lines are processed from left to right. The appearance of a postconditional expression (see Chapter 6) will

defer action of a Command until it has been evaluated true, or ignore the remainder of that Command, but the left-to-right sequence is maintained. In the next example, assume A=0 and B=7

```
SET:A B=B*A WRITE "THE VALUE OF B IS ",B
```

In this example, since A=0, the SET B=B*A is ignored. However, the postconditional limitation does not affect the remainder of the Command Line, and the MUMPS interpreter would write out THE VALUE OF B IS 7, since B was not changed. If A had been any other value besides 0, it would have been considered true (i.e. not zero), and B would have been changed accordingly.

IF Command and $TEST Special Variable

One of the greatest capabilities of today's computers is the ability to branch conditionally, that is, to perform certain actions only under certain conditions. MUMPS has several ways of exploiting this characteristic. We have already seen the $SELECT and postconditional applications of conditional execution. We now turn to the use of the IF Command. In MUMPS, the range of the IF command is the Command line to the right of the IF. For example, suppose X=5,Y=3, and the following Command Line appears

```
IF X<Y SET Z=Y#X WRITE "Z = ",Z
```

Upon execution of this Command Line, MUMPS will first determine whether the argument of the IF command is true. Since, in this case, it is not true, the SET and WRITE commands will be ignored. On the other hand, if we had the following Command Line (using the same variables)

```
IF X>Y SET X=Y WRITE "X HAS BEEN RESET TO ",X
```

The IF argument in this case evaluates as true, and the SET and WRITE statements are executed. Remember that the Commands affected by the IF evaluation must lie on the same Command Line. The next Command Line in sequence will be executed normally. There is, however, one change to a MUMPS special variable, $TEST. Upon the evaluation of an IF argument, $TEST is set to 0 or 1, depending on whether the argument evaluates as false (0) or true (1).

$TEST is useful in extending the scope of the IF argument evaluation. Suppose that we had a number of actions to take depending on the true evaluation of an IF argument. If these actions cannot all be placed on one Command Line, then, without the $TEST special variable, it would be necessary to repeat the IF argument on each successive Command Line. Consider the following example, assuming AGE=19, RES="CALIFORNIA":

```
IF (AGE>18)&(AGE<21)&(RES="CALIFORNIA") WRITE !
IF  WRITE "SINCE YOUR AGE IS ",AGE
IF  WRITE "AND YOU LIVE IN ",RES
IF  WRITE "YOU MAY VOTE, BUT YOU CAN'T BUY BEER.",!!
```

(Please note that the last two characters on the fourth line are format controls, not emphatic remarks). In this example, the IF argument in the first line is tested, found to be true, and $TEST is set to 1 accordingly. Then the remainder of that line is executed. Upon encountering the second line, the IF Command is encountered followed by <u>two</u> spaces. The second space signifies the use of an IF without any arguments ("argumentless IF"). In this situation, MUMPS will examine the value of $TEST (set true on the previous line), and, finding it true, will execute the remainder of the Command Line. Similarly, the next two lines will also be executed.

<u>ELSE Command</u>

Sometimes it is desirable to do <u>either</u> one action or another, depending on the evaluation of an IF argument. The argumentless ELSE command allows us to do the exact inverse of the argumentless IF statement. In other words, ELSE examines $TEST and, if its value is zero, the remainder of the Command line is executed. We can therefore rewrite the previous example as follows:

```
IF (AGE>18)&(AGE<21)&RES="CALIFORNIA") WRITE !
IF  WRITE "SINCE YOUR AGE IS ",AGE
IF  WRITE "AND YOU LIVE IN ",RES
IF  WRITE "YOU MAY VOTE, BUT YOU CAN'T BUY BEER",!!
ELSE  WRITE "YOU MAY VOTE AND BUY ALL THE BEER YOU CAN AFFORD"
```

(We are assuming that California is the only state with these restrictions). This example will execute the first three Command Lines only if the IF argument is true; otherwise, it will execute only the fourth Command Line.

There is no argumented form of the ELSE command, since the same effect can be achieved by simply inverting the sense of the IF argument.

Summary

In this chapter we have introduced the concept of a Command Line, identifying its principal components and illustrating the multiple commands that may be executed on one line. The conditional execution of the MUMPS language IF and ELSE commands are also illustrated to demonstrate the limited scope of those command to single Command Lines.

Chapter 8

Routine Structure

Having established the nature of Command Lines in the previous chapter, we now turn to the next level of organization for MUMPS code - the Routine. A Routine is a collection of Command Lines, all of which are associated with a single name, the name of the Routine. This name, which follows the same conventions as a variable name is used for reference purposes: to store or retrieve the Routine, or to access it from other Routines using Commands discussed in the next chapter. By general convention, MUMPS routines usually begin with a line which consists of the label which is the routine name followed by a comment giving its author, date of creation and last update.

The body of the routine consists of a series of command lines, which may or may not be labelled. Typically, a Routine will contain several lines with labels interspersed with others having none.

The end of a Routine is marked (on most systems) by two ASCII characters, the Carriage Return (CR) and Form Feed (FF) as illustrated in Table 1. MUMPS makes use of this convention to determine that the end of a Routine has been reached.

Statements are usually executed in the sequence in which they appear within a routine, and we have not yet encountered any Command that would allow us to do anything other than skip a Command (postconditionals) or Command Line (the IF and ELSE Commands). Sometimes, however, it is necessary to construct a Routine in which the sequence of execution will not be in the sequence in which the lines are stored. The command line label is the identifier used to specify the beginning of the new sequence of execution. The specific Commands permitting transfer of control within a Routine are DO and GOTO, discussed in the next section of this chapter.

GOTO Command

In some problems, it is necessary to devise branches which bypass blocks of instructions. Although one way to accomplish some

such branches might be through the extended use of IF commands such as those described in the previous chapter, a more convenient way of skipping large blocks of code is by using the GOTO command. GOTO label instructs the MUMPS interpreter to execute, as its next step, the Command Line associated with the label referenced. An example of such branching logic might be the following:

```
     READ !,"PLEASE TYPE YOUR SEX (M OR F)",SEX IF SEX'["F" GOTO MALE

     READ !,"HAVE YOU EVER EXPERIENCED AMENORRHEA (Y/N)",ANS1

     READ !,"ARE YOUR MENSTRUAL PERIODS REGULAR (Y/N)",ANS2 GOTO
CONT1

MALE READ !,"HAVE YOU BEEN CIRCUMCISED (Y/N)?",ANS3

     READ !,"HAVE YOU EVER HAD DISCOMFORT OF THE PROSTATE
(Y/N)?",ANS4

CONT1 READ "DO YOU CURRENTLY SMOKE TOBACCO (Y/N)?",ANS5

     IF ANS5["Y" READ "RARELY, MODERATELY, or HEAVILY (R/M,H)?",ANS6
```

It is easy to see that much more complicated branching questionnaires can be constructed using this technique. Incidentally, one of the more common uses of the MUMPS language is in the area of questionnaires, especially in obtaining patient histories and in education.

DO and QUIT Commands

The DO label also transfers execution control to the referenced line. However, in the case of the DO Command, control is eventually transferred back to the next Command after the DO statement. There are two ways in which the transfer of control can terminate, returning control to the point at which it was interrupted by the DO statement. The first is when the list of Command Lines remaining in the Routine is exhausted; the second is when a QUIT Command is encountered in the referenced sequence of instructions. An example of the former is given by the following miniature Routine:

```
PCTACTIV ;RFW WRITTEN 7/78, UPDATED 9/78

     ;GETS HOURS SPENT IN VARIOUS ACTIVITIES, CONVERTS TO PERCENT

     WRITE "WE WOULD LIKE INFORMATION ON YOUR AVERAGE DAILY LIFE."
```

```
     WRITE !,"PLEASE TYPE THE NUMBERS OF HOURS SPENT DAILY IN"
     WRITE !,"EACH OF THE FOLLOWING ACTIVITIES",!
START READ !,"SLEEPING ",V DO PCT SET SLPCT=Y
     READ !,"READING OR STUDYING ",V DO PCT SET RDPCT=Y
     READ !,"EXERCISING ",V DO PCT SET EXPCT=Y
     READ !,"COMMUTING/TRAVELLING ",V DO PCT SET COMPCT=Y
     READ !,"EATING AND RELAXING ",V DO PCT SET RLXPCT=Y
     SET TOTAL=SLPCT+RDPCT+EXPCT+COMPCT+RLXPCT
     IF TOTAL>.9 GOTO ENOUGH
     ELSE READ !,"WHAT ELSE DO YOU DO?",OTHER
ENOUGH  IF  TOTAL >1.1 WRITE "TOO MANY HOURS. RE-ENTER DATA" GOTO START
     WRITE !,"THANK YOU" GOTO FINISH
PCT SET Y=V/24
FINISH ;END OF ROUTINE(CR FF)
```

Although this is a somewhat contrived example, it illustrates the repeated use of a segment of code that can be invoked by the DO label Command. Note that the DO Command goes to the bottom of the Routine, executes code until it sees the CR FF characters (which do not print on the terminal), then returns to its origin to continue sequential execution. Since the line labelled FINISH has no Command, no action is performed when the interpreter reaches this line.

Another way to accomplish the same task would be to rewrite the line labelled PCT as follows:

```
PCT SET Y=V/24 QUIT
```

When the MUMPS interpreter encounters the QUIT Command, it returns to the place where the DO Command appeared. If we had such a QUIT statement, we could have several such subroutine Command Lines at the end of such a Routine.

Postconditionals on Arguments

In Chapter 6 we saw how commands could be conditionally executed through the use of truth-valued expressions called postconditionals. Certain commands (GOTO, DO and EXECUTE) also permit postconditionals to be placed on their arguments. Whie a false command postconditional prevents execution of the whole command (with all of its arguments), a false posconditional on an argument only prevents execution of the command on that argument. Further arguments are executed normally. The following lines ask for verification of entered data and transfers control to the appropriate label:

```
CHECK READ !,"DATA OK (Y/N)? ",X
     GOTO DONE:X="Y",ENTER:X="N" WRITE "?" GOTO CHECK
```

Note that if the response is incorrect, the program types a question mark and repeats the question.

HALT and HANG Commands

Frequently, instructions that accompany cars, appliances and computer languages leave till quite late in the manual the magic wand required to stop the machine or program. This PRIMER is no exception. We will now provide you with that essential piece of information.

HALT is a Command that has no arguments and does precisely that. It stops all execution and returns control to the operating system. HALT can have no arguments, nor can any Commands following HALT on the same Command Line be executed(unless HALT has a postconditional). It is a comforting feeling for the programmer to know that there is a way of escaping (short of unplugging the computer) from the situation at hand.

Although HANG may sound somewhat more drastic even than HALT, it is in fact a far more innocuous Command. HANG permits the user to suspend operations for a specified number of seconds, then continue. The MUMPS interpreter will silently count the appropriate time, then proceed to execute the next statement. Like most MUMPS statements, HANG may be modified with a postconditional statement. To illustrate, suppose that we knew that the user of our Routine might

use either a printing or a CRT terminal. If we are sending many lines to the terminal, we might wish to delay at the end of a screen of output, allowing the user to read the material generated before it is replaced with the next page of information. In such a situation, we might wish to establish a variable DEV, and, at the beginning of the execution of the routine, insert the following statement:

```
READ !,"ARE YOU USING A PRINTING TERMINAL (Y/N)?",DEV
```

Later, when we came to the part of the Routine that was doing the repetitive printing, we might insert the following Command Line:

```
HANG:DEV["N" 10 ....(Continued Commands)
```

This Command would determine whether the user had entered a Y(ES) or N(O), and, if he has said no to using a printing terminal, execution would pause for ten seconds prior to continuing with the next Command. Note that we used the contains operator DEV["N", rather than saying DEV=N, since it frequently happens that users will write out either YES or NO, ignoring instructions such as those printed in the READ statement.

Summary

We have now covered all of the elements required to write a complete MUMPS Routine (although we will continue to introduce a few more Commands that enlarge the possibilities within such a Routine). In this chapter we learned that Routines begin with an identifying name, consist of a series of Command Lines which may or may not be labelled, and terminate with a Carriage Return and Form Feed (CR FF). We also learned how to transfer control within a Routine, illustrating both the GOTO and the DO forms for branching.

In the next chapter, we will consider ways in which we can access Routines from external locations. This capability will complete our library of tools for executing MUMPS instructions, and we can then turn to file management considerations of the language.

Chapter 9

Program Structure

Routines are collections of Command Lines which are stored by the MUMPS interpreter in such a way that they can be retrieved and used as a unit. Routines are given names, and by convention the first line of each Routine contains its name, date of creation and the author's name or initials.

The final step in creating a computer solution to an information handling problem is to link Routines together so as to form Programs. By use of the DO/QUIT combination, it is possible to create a modular solution involving a series of Routines, each solving a limited portion of the problem. GOTO may also be used for transfer of control between Routines, though it offers somewhat less rigorous control of the flow of control between Routines.

GOTO and DO/QUIT Commands Used with Routines

There are two ways to access Routines that have been previously stored. They involve the use of the GOTO and DO Commands described in the previous chapter, with one additional feature: the name of a Routine is identified by a caret (^, which prints on some terminals as an up-arrow). The action of GOTO ^Routinename is the same as if one used the same Command with a label within a Routine. The transfer of control is to the top of the Routine indicated, and each Command in that Routine is executed according to the statements encountered. Suppose, for example, that we were processing a series of invoices. Such a task could be divided into separate tasks associated with identifying the originator of the invoice, the goods or services referenced, the amount, the unit to be charged, payment of the invoice, and summary statistics of the cumulative results of the cash flow generated by the invoices. Several of these tasks might logically be placed together; others are readily separated, and the summary information could not, in fact, be produced until the final invoice for that time period had been processed. It is customary in most computer solutions of such problems to subdivide the tasks into smaller subunits. The Routine is such a unit in MUMPS. A hypothetical sequence of such Routines might be partially illustrated as follows:

```
ORIGIN  ;CREATED 9/1/76 J.MCKAY
     ;IDENTIFY BILLING AGENT,PRODUCT AND AMOUNT
     ...
     ...GOTO ^DEBIT
     ...
DEBIT ;CREATED 9/15/76 J.MCKAY
     ;VALIDATE ACCOUNT NUMBER TO BE CHARGED, DEBIT ACCOUNT
     ...
     ...GOTO ^PAYMENT
     ...
PAYMENT ;CREATED 9/6/76 J.MCKAY
     ;WRITE CHECK ON SPECIAL PRINTER;TYPE ENVELOPE LABEL
     ...
     ...IF END GOTO ^SUMMARY
     GOTO ^ORIGIN
SUMMARY; CREATED 9/20/76 J.MCKAY
```

This sequence would continue to process invoices according to the sequence specified until the condition "END" had been set true, at which time the program would transfer to the Routine named SUMMARY.

As we have noted before, this example does not illustrate good programming practices (GOTO used in this manner is contrary to the usual conventions of structured programming). However, it does indicate one way in which such a problem might be subdivided between Routines in a complex program.

A more acceptable solution for many situations requiring transfer of control between Routines is the use of the DO/QUIT sequence. In this approach, we use the DO Command with the name of the Routine preceded by a caret (^); execution of this Command will

transfer control to the top of the named Routine. Once in that Routine, Commands are processed until the end of the Routine is reached or until a QUIT appears as a Command within the Routine. A QUIT is an alternative way of returning to the Routine in which the subsidiary code was invoked. This combination of DO and QUIT presents an effective way of transferring control in a manner that is easier to trace and maintain. The same Routines illustrated in the previous example might contain the following sequence of statements:

```
MASTER ;CREATED 9/1/76 J. MCKAY

     ;MASTER CONTROL PROGRAM FOR PROCESSING INVOICE

BEGIN ...
          IF END DO SUMMARY QUIT
          DO ^ORIGIN
          DO ^DEBIT
          DO ^PAYMENT
          GOTO BEGIN
          ...
ORIGIN ;CREATED 9/15/76 J.MCKAY
          ... QUIT
DEBIT ;CREATED 9/10/76 J.MCKAY
          ...QUIT
SUMMARY ;CREATED 9/20/76 J.MCKAY
     ...QUIT
```

In this sequence, the main program (MASTER) contains all the principal program flow for the entire program. Changes in the basic sequence of major steps performed can be readily accomplished in this Routine without affecting the details of processing each subactivity. Note that a Routine may contain more than one QUIT, allowing the programmer to select alternative branches for execution and return.

MUMPS does not require that all Routines to be used by a given

program be in the computer's memory at the start of execution. Instead, the interpreter will load the Routines as they are required. In some implementations the Routines are accessed from disk each time they are invoked. Other implementations maintain buffer areas for Routines, thereby reducing the number of disk accesses. The net effect to the user is that he may design programs that are arbitrarily large or complex, provided that each Routine is small enough to be loaded as a single unit. This feature of MUMPS tends to promote modularity, a program characteristic that often facilitates modification and maintenance of complex programs.

Labels and Routinenames can also be used together in the arguments of GOTO and DO. GOTO ENTER^ABC transfers control to routine ABC, starting at the line labelled ENTER rather than at the beginning of the routine. If it had been more convenient to place the code for debits and payments in the same routine in the example given above, we might have written Master as:

```
MASTER ;CREATED 9/1/76 J. MCKAY

     ;MASTER ...

BEGIN IF END DO SUMMARY QUIT

     DO^ORIGIN,DEBIT^ACCNT,PAYMENT^ACCNT GOTO BEGIN
```

Here, DEBIT and PAYMENT are labels in the routine ACCNT, and we have used a multiargument form of the DO command.

Packages

Thus far, in reviewing MUMPS executable statements, we have progressed from Commands through Command Lines to Routines and Programs. With this hierarchy it is possible to solve a large variety of problems involving manipulation of complicated data sets. One further naming convention is also used in MUMPS: that of a Package. As one might suspect, a Package is a collection of Programs, each of which contains a series of Routines. Programs are generally groups of Routines addressing a single process, whereas Packages may involve a wider range of Programs that, taken together, serve a larger administrative or technical goal. For example, a hospital laboratory has sections of clinical chemistry, microbiology, and hematology. Each of these units might be served by a Program. The entire laboratory would have Programs for the services listed above as well as those for specimen collection, reporting results, inventory, accounting, patient identification, etc. A company

marketing such a total system would tend, in MUMPS circles, to refer to this agglomeration of Programs as a Package. MUMPS Packages exist to solve the specific problems stated above, to support television and radio scheduling, commercial account systems and a host of application packages in a wide range of disciplines.

Summary

With the completion of this chapter, we have covered all of the basic elements necessary to solve simple problems using the MUMPS language. The remaining chapters introduce concepts that broaden the extent of these applications by describing ways in which data may be classified, stored and retrieved in the MUMPS environment. Since one of the unique features of MUMPS is its file management, this capability is a particularly important concept in the effective use of the language. Completion of the next three chapters will, however, conclude the introduction to the chief programming aspects of the language. The remainder of the text is devoted to some special extensions to the language that are somewhat beyond the introductory level and for that reason have been placed at the end of the PRIMER.

Chapter 10

Local Arrays

Introduction

Until now, we have only been concerned with variables which are known as simple variables, those which have a one-to-one association between the variable name and its value. Simple variables serve us well in many situations, but sometimes we need a way to manipulate a larger collection of data, such as a list of employee names. While it would certainly be possible to devise a new simple variable name for each employee number of name, doing so would make programming output an extremely awkward process. MUMPS, like most other programming languages, solves this problem by including, as part of the language, arrays.

An Array is a type of variable that has a simple name, but it may also have many values associated with it. When it is necessary to make reference to one individual value out of the collection, a subscript is added to the name, serving to identify the particular value desired. Since most computer terminals lack the ability to print subscripts one-half line below the regular text (such as O_2), MUMPS (like most languages) encloses the subscript in parentheses immediately after the name, such as A(1). With subscripted variables, our employee list problem can be greatly simplified. We could store the first number in A(1), the second in A(2), and so on until we had identified each employee. To further extend this concept, we could go beyond the use of a constant as the subscript, since MUMPS will allow us to use an expression in the place of a constant as a subscript. Using this fact, we might read our employee number into an array from a terminal with the following portion of a routine:

```
     SET N=0 ;N WILL COUNT NUMBER OF EMPLOYEES
RD READ !,"NEXT EMPLOYEE NUMBER?(NULL WILL TERMINATE)",NO
     GOTO:NO="" DONE ;GO TO 'DONE' IF NULL ENTERED
     SET N=N+1,EMPNOS(N)=NO GOTO RD ;LOOP BACK FOR NEXT NUMBER
DONE WRITE !!,"YOU ENTERED ",N," EMPLOYEE NUMBERS."
```

This type of programming, utilizing subscripted arrays that can be referenced by other variables or even expressions,greatly simplifies the job of the programmer. We will return to the concept of subscripting in MUMPS to examine still other features later in this chapter.

FOR Command

Once an array is loaded with values, the values can be manipulated or examined at will. In order to manipulate the values of an array in sequence, MUMPS provides the FOR command, which allows the programmer to execute a command line repetitively. In its most general form, the FOR command has the following syntax:

```
FOR INDEX=P1[,P2,...] (the rest of this line will be executed)
```

where INDEX is a variable name, and P1,P2,... are forparameters, as defined below.

The forparameters are used to successively assign values to the index variable. Each time a new value is assigned to the index variable, the rest of the command line is executed once. This repetition continues until all forparameters have been exhausted or until the command is terminated in some other way (e.g. by an QUIT or GOTO statement later on the same line).

Let us illustrate this concept with a series of examples. The simplest forparameter consists of a single expression. In this case, the index variable is assigned the value of the expression, the line executed once, and the FOR command continues on to the next forparameter. For example, the lines

```
SET S=""
FOR X="A","B","C" SET S=S_X WRITE !,S
WRITE !,"DONE"
```

would result in the following output:

A

AB

ABC

DONE

Note that X, the index variable, was successively assigned the values "A","B", and "C", as specified by the forparemeters and that when all the forparameters ,had been processed, execution of the command line containing the FOR command was terminated, and control passed to the next statement.

The remaining two forms of the forparameter require numeric expressions, as they specify iterative execution of the command line. The first form is

numexpr1:numexpr2

In this case, the index variable is first assigned the value of numexpr1, and the line is executed. The value of numexpr2 is then added to the index variable, and the line is executed again. This process repeats endlessly in the form written above, unless the command line is terminated in some other way (see QUIT in FOR, below). As an example, execution of the following line will cause an endless list of even numbers to be printed on the output device:

```
FOR I=0:2 WRITE !,I
```

producing

0

2

4

...

until the process is stopped by some external interruption (such as disconnecting the terminal or executing an implementation specific command to abort a program). We will see situations where this form

of the FOR command may indeed serve a useful purpose, though it certainly does not in the example shown above.

The third form of the forparamter is similar, except that it does not specify an endless loop. In this final form, a third numeric expression is added which specifies the limit value to cause termination of execution. Referring back to our earlier example of employee numbers, the following short Routine will examine the array EMPNOS (which contains N entries according to our previous example), to see if NO is a valid number. If it is, I will return "pointing to" the employee number in EMPNOs, otherwise I will return as a zero.

```
CKNO FOR I=1:1:N IF EMPNOS(I)=NO GO TO DONE

     SET I=0 QUIT

DONE QUIT
```

This program will loop until the value assigned to NO has either matched an employee number or all employee numbers have been unsuccessfully tested. Note the third numeric expression (N), which provides an upper limit to the value to be assigned to I. In this example, I begins with 1, and the first employee's number is compared with NO. If they are equal, control is transferred to the line labelled DONE; otherwise the line is completed and execution returns to the forparameter. The second expression in the forparameter is added to I and the next employee number is checked. If, however, adding 1 to the last value of I would cause it to exceed the limiting number N, control transfers to the next line, setting I to zero and QUITting, indicating that NO did not match any employee number in the file.

In both forms of the forparameter that require numeric expressions, the increment value may be negative as well as positive. This option makes little difference in the case of the endless loop example given above, but in a three expression form, the significance of the last term is reversed. In such a case (i.e., a negative second expression), this incrementing value is repeatedly subtracted from the value of the index variable until the point at which subtraction of one more value would result in the index variable being less than the limiting variable. Thus,

```
FOR I=9:-2:4
```

would execute the line with I have successive values of 9,7, and 5. Subtracting 2 from 5 would produce 3, a number smaller than the limiting variable, 4. If the initial value of the index variable is greater than the limit (for a positive increment) or less than the limit (for a negative increment), the line is not executed. Thus both

```
FOR I=10:1:5 <line of commands>
```

and

```
FOR I=2:-1:4 <line of commands>
```

will result in the remaining portions of those lines being ignored. The value of I, however, will be affected, being set to 10 and 2, respectively.

The various forms of forparameters may be mixed within the same FOR command. For example

```
FOR I=0,2:1:5,8,11,-1,15:-1:13 WRITE I,","
```

would result in the output

```
0,2,3,4,5,8,11,-1,15,14,13,
```

It is also possible to have more than FOR command in the same line. Since the range of a FOR command includes the rest of the command line, it is obvious that successive FOR commands will be nested one inside the other. Hence the command lines

```
SET S=""
FOR I=1:1:3 SET S_"A" for J=1:1:I SET S=S_"B"
WRITE !,S
```

results in the single output line

```
ABABBABBB
```

the outer FOR loop, controlled by the index variable I, includes the inner FOR, controlled by J within its range. Thus the inner FOR is executed to completion each time the outer loop is executed only once, resulting in the illustrated output.

MUMPS also permits the control variable to be modified with the executed command line. For example:

```
FOR I=1:2:10 WRITE I SET I=I-1
```

would result in

```
12345678910
```

This capability should be used with great caution, since it may easily lead to unexpected results.

GOTO and QUIT in FOR

In our example of checking for employee numbers, we saw one way to terminate a FOR command prior to completion of all its forparameters, i.e., by executing a GOTO command within the range of the FOR command. Since the GOTO command terminates execution of the entire command line it naturally terminates any FOR commands on that line. The QUIT command may also be used to terminate a FOR command, but it operates in a slightly different way. QUIT, when executed within the range of a FOR command, causes termination of the innermost FOR, containing the QUIT. By modifying slightly the previous example, we can illustrate this concept:

```
SET S=""
FOR I=1:1:3 SET S=S_"A" FOR J=1:1 QUIT:J>I SET S=S_"B"
WRITE !,S
```

when executed, results in

ABABBABBB

Note that the QUIT terminates the endless loop specified by the inner FOR command as soon as J exceeds the value of I. This QUIT command does not, however, terminate the execution of the outer loop controlled by the index variable I.

More About Arrays

Returning to the concept of Arrays, we now see that the FOR command gives us a powerful tool for operating on arrays. There are some additional points, however, that should be discussed with respect to arrays in the MUMPS language. In most programming languages, if you wish to assign values to items 1 and 10 of a given array, you must provide space for all intervening values as well (i.e. 2 through 9). Such arrays are usually declared ahead of time, warning the computer that space must be reserved for this array and specifying how many spaces to reserve. Such an array is referred to as a "dense" array, meaning that every subscripted value is provided for in the storage assigned to the program.

MUMPS differs from this approach in two important ways. First, there is no requirement that arrays be "dense"; instead, MUMPS permits "sparse" arrays in which only those variables that have been assigned values occupy space in the symbol table. Second, it is not necessary to specify this storage space ahead of time. MUMPS will, upon encountering a new variable, find the space and assign the designated value. The use of sparse arrays thus conserves space and eliminates the need for declaration statements reserving space (which may never be completely used). In a more sophisticated sense, it also allows for the potential selection of subscripts that might have more meaning than a simple numeric sequence. Returning once again to our example of employee numbers, it is possible to use the employee number itself as a subscript, so that data such as name, address and salary could be stored in arrays referenced by employee number key.

Multiple Subscripts

All of the examples used up to this point have used only a single subscript, to allow us to focus on other aspects of the language. MUMPS does, however, permit multiple subscripts. In fact, arrays may have any desired number of subscripts (subject only to limitations of line length). For example, a single array may be defined to contain employee names and address according to the

following scheme:

EMPNA(EMPNO,1) will contain the name

EMPNA(EMPNO,2) will contain the street address

EMPNA(EMPNO,3) will contain the city and state of residence

EMPNA(EMPNO,4) will contain the ZIPCODE.

In this example, EMPNA is a two-dimensional array (a matrix) with the first dimension keyed to the employee number and the second dimension containing four additional data items for each employee. Other, far more complex, dimensioned arrays could be devised, allowing for entry of multiple academic degrees received, dates and types of jobs held with the company, etc.

Permissible Subscript Values

Standard MUMPS makes no restrictions on the allowable values of subscripts, requiring only that they be expressions. The portability requirements of the language do, however, restrict subscripts to nonnegative integers. Many MUMPS implementations adhere to this restriction, though some do permit negative numbers, non-integers or even strings as subscripts. For details on your system, refer to your local documentation. For a discussion of the possible extension of the language in this respect, see Chapter 16.

Placement of Data Values

Data values may be placed anywhere within a MUMPS array. Values are not limited to the "lowest" subscript level. Often, documentation is placed at "higher" levels to aid in the use of the files. Consider the following alternate organization of our employee file:

Variable	Value
EMPNA(EMPNO,2)	employee department

etc. In this approach, the file is to some extend self-documenting, and a user could, by examining the file at the top level, determine

its general nature.

KILL as used with Arrays

The KILL command (Chapter 6) may also be used to delete arrays. When used without arguments, KILL deletes all variables, subscripted or otherwise. The selective KILL permits you not only to eliminate specific variables, but even specific subscripted values within such arrays. Thus, KILL EMPNOS(3) will eliminate only that value, whereas KILL EMPNOS will erase the entire array. Using the selective form of KILL, you may exclude an entire array from being killed (KILL (EMPNOS)), but you may not exclude only one element of such an array (e.g., KILL (EMPNOS(3)). Such a statement would be treated as an error.

There are a few additional points that should be made with respect to the use of the KILL command on variables. Any name may, at the same time, have both subscripted and unsubscripted forms, and the number of dimensions for any given subscripted reference may differ from other reference identified by the same general name. Thus A, A(1), A(1,2) A(9,2), and A(27,3,4) may all exist simultaneously within a MUMPS environment. In this situation, executing KILL A will kill both the subscripted and the unsubscripted forms. Similarly, A(1) and A(1,2) may both be deleted by executing the command KILL A(1), which will KILL the referenced variable and all its "descendants", i.e., those which are hierarchically below the named variable. However, executing a KILL A(1,2) will not affect A(1).

Array Manipulation Functions

MUMPS provides two functions to manipulate sparse arrays. Since referencing an undefined value in MUMPS is an error, you must be sure that, for instance, A(10) exists before you use it. The $DATA function will help you ascertain the status of any variable. The argument to $DATA is a variable name, including subscripts if desired. $DATA returns the values as shown in the following table:

Argument	Value returned	Meaning
variable	0	undefined, no descendants
variable	1	defined, no descendants
variable	10	undefined, w. descendants
variable	11	defined, w. descendants

In this table, descendant means a subscripted variable with more subscripts than the argument variable whose leading subscripts, if any, are identical to those of the argument variable. For example, in our example of the previous paragraph, A(1,2) is a descendant of A(1). By the same logic, A(1,2,4,9) is a descendant of A(1,2), etc. Since KILLing a descendant has no effect on the ascendant (or parent), if A(1) and A(1,2) are the only variables of A defined, and A(1,2) is KILLed, $DATA A(1) will still return 11, since there were at one time descendants of this variable. On the other hand, if we had a variable A(3,7) but no variable A(3), then the $DATA of A(3) would give the result 10, since A(3) had not been defined but lower variables do exist. The $DATA is defined in this manner regardless of the extensions to subscripts permitted in some implementations.

Given that you can determine, through use of the $DATA, whether descendants of a subscripted variable are present (i.e. if the value returned is either 10 or 11), the next logical question is how to find out the subscripted values for those descendants (assuming that you don't already know them). The $NEXT function may be used to identify the next subscript in sequence at the given level. The argument to $NEXT must be a subscripted variable, with the last subscript being an integer. The value -1 for the last subscript is reserved as a special signal, indicating that you want to know the first subscript at that level. $NEXT returns an integer value which is either the next higher subscript with a defined value or with descendants, or else it returns a -1, indicating that there are no more subscripts at that level.

This seemingly complex function can be understood best through a specific example. Suppose that we have defined the following subscripted values for the variable X:

X(1), X(1,2), X(1,4) X(1,5,6), and X(2)

If we apply $NEXT to each of these variables in turn, we obtain the following results:

Variable	$NEXT(Variable)
X(-1)	1
X(1)	2
X(2)	-1
X(1,-1)	2
X(1,2)	4
X(1,4)	5
X(1,5)	-1
X(1,5,-1)	6
X(1,5,6)	-1

Careful study of this example together with the preceding paragraph should elucidate most alternatives of the $NEXT function.

Used in combination, $DATA and $NEXT can help you define the profile of an array, as shown in the diagram 10.1 accomanying this chapter.

Summary

This chapter has introduced the concept of subscripted "sparse" arrays that characterize MUMPS. It has also shown that, despite the fact that we cannot tell beforehand the specific subscripts that may appear in a given variable (since it is not declared beforehand and since it is sparse), use of the functions $DATA and $NEXT will allow us to map an array completely.

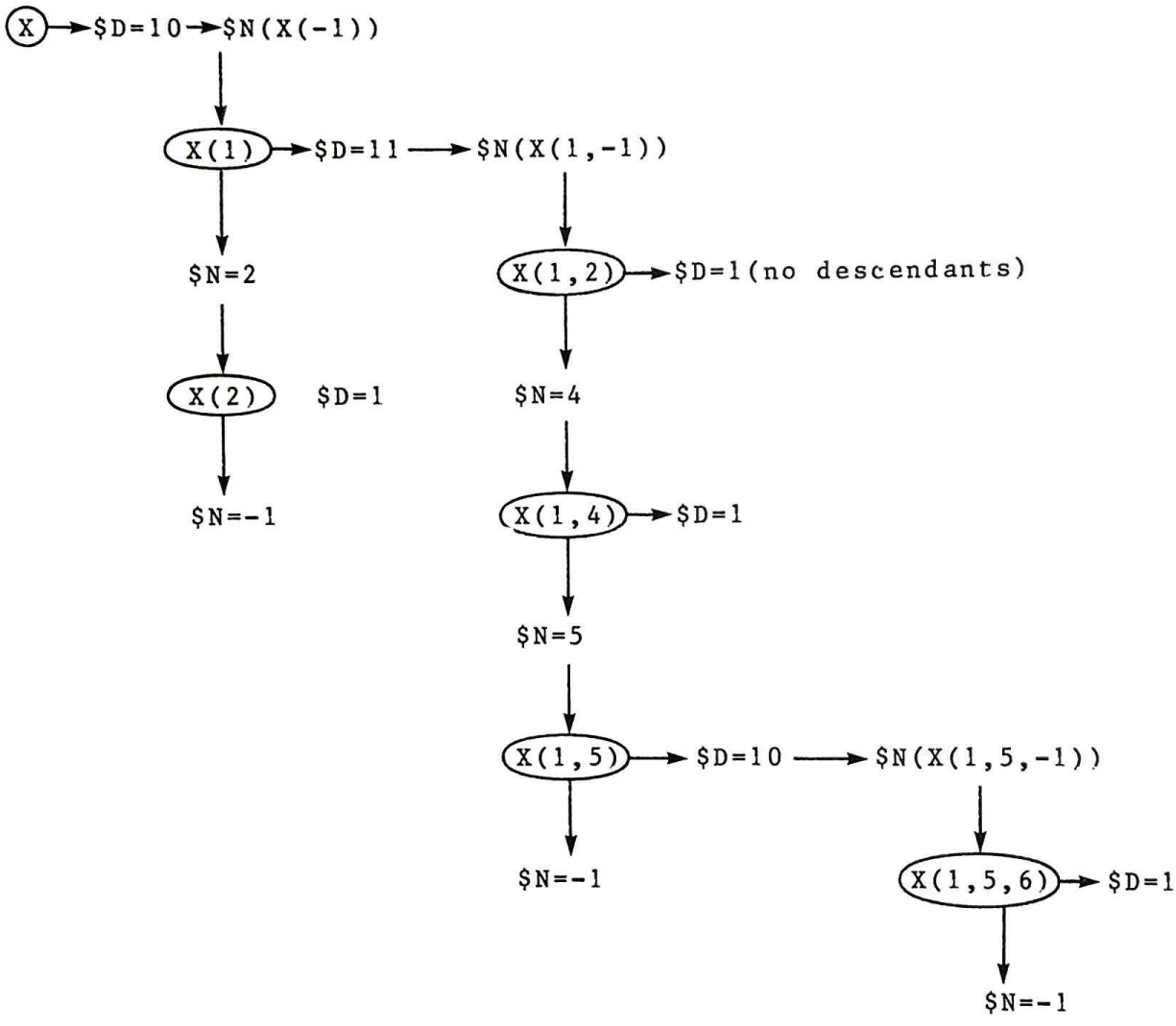

Fig. 10.1 Determination of an Array Profile Using $NEXT and $DATA.

Chapter 11

Advanced Input/Output

In Chapter 6 we discussed use of the READ and WRITE Commands on the terminal. Besides a terminal, MUMPS allows programmers to perform normal input/output (I/O) operations on other devices, such as line printers, tape drives and the like. The same READ and WRITE commands are used, with some extended syntax as described below.

OPEN Command

Before any device or file may be referenced in an I/O operation, the device or file must be assigned to the program. Since the MUMPS system usually supports a number of simultaneous users, this provision is required to permit sharing of resources in an orderly manner. Assignment of a device to a user is accomplished by executing the OPEN command. Each argument of the OPEN command consists of three parts, only the first of which is required:

1) An expression which serves to identify the device or file to be OPENed;

2) one or more expressions, called deviceparameters that provide information necessary to describe the device or file; and

3) an expression that indicates the maximum amount of time the program should spend attempting to obtain the device or file. This expression is referred to as the timeout.

Because the nature of I/O operations are highly dependent on the particular device and implementation of MUMPS, the ANSI Standard does not define the interpretation either of the identifier expression or the deviceparameter expression. For specific details on your installation, consult your system documentation. In this section, we will focus on the use of the OPEN command, ignoring the deviceparameter details.

When a progam issues an OPEN command, MUMPS attempts to give that program exclusive "ownership" of the requested item, thereby

preventing two users from accessing the same devices simultaneously. Upon encountering an OPEN request, execution of further code in the user's instruction sequence is suspended until ownership is obtained. If there is a timeout specified (the third part of the OPEN argument), control will return to the user's instructions if that time limit is exceeded. If ownership is obtained within the designated time, the special variable $T is set true, indicating a successful OPEN. If ownership cannot be obtained within the time limit (i.e., some other user already has the device and it is therefore not available), the $T special variable is set false. The user can therefore test this variable to determine if he has obtained ownership. If no timeout is present, $T is not affected. However, the OPEN will only return if ownership has been obtained.

To illustrate, let us assume that in our implementation, device 0 is the user's terminal, device 1 is a line printer, and device 2 is a card reader. It is not necessary to OPEN the terminal in most systems, but if our program wishes to use the line printer and card reader, these devices (which could be used by others) must be OPENed. The command to open them might be

```
OPEN 1:dl,2:dr
```

where dl and dr are the deviceparameters for the line printer and card reader, respectively. In this example, no timeout was specified. If we were willing to wait indefinitely for the line printer, but wanted to wait no more than one minute for access to the card reader, the modified command line would be

```
OPEN 1:dl,2:dr:60
```

where the final number 60 denotes sixty seconds timeout. After this time, $T could be tested to see if the reader had been assigned to the user.

USE Command

Once a device or file has been OPENed, we must indicate which device we wish to reference with our READ and WRITE statements. Since the user always communicates with MUMPS via a terminal, OPENing another device creates an ambiguous situation unless a way is found to identify the specific device intended. We accomplish this task through the concept of the current device, which is the single device referenced at any given time by READ and WRITE commands. Initially,

the current device is generally the terminal. When the program wishes to change the current device, it executes a USE command. Each argument of the USE command will identify the device to be referenced, possibly followed by deviceparameters, which once again are implementation specific. As each argument is executed, the file or device becomes the current device. Obviously, it makes no sense to have multiple arguments to the USE command, since only the last one will have any effect on subsequent actions (except as deviceparameters may have side effects). The special variable $IO is set in conjunction with the USE command to identify which device is currently designated as the current device. Since execution of a USE implies that I/O operations (via READ and WRITE) will be sent to the device, the device argument must be currently owned (by a previous OPEN command).

To continue the example from above, executing a

```
USE 1
```

command would cause all READ and WRITE commands following its appearance to reference the line printer. $IO would also be set to 1. This condition would remain in effect until another USE command is executed, identifying some other device. Since a READ command has no meaning when we are addressing a line printer, we would have to take care that we did not include READs while the line printer is the current device.

For each file or device, separate $X and $Y special variables are maintained (see Chapter 6 for definitions of their meaning). The format controls (!,#, and ?) are also defined for all files or devices, though their exact effect will vary and must be defined individually by the implementor.

CLOSE Command

Once all I/O instructions involving a special device have been completed, the user should relinquish ownership of the device. Release of a device is accomplished through the CLOSE command which, like OPEN and use, has an argument identifying each device that is to be CLOSEd. Deviceparameters may also be appended as specified by the implementor. If ownership of the current device is relinquished, the normal action is to return to a default device, usually the terminal. The Standard does not, however, require this action and some systems may differ slightly in the default action.

Extensions of READ and WRITE

The final topic to be addressed in this chapter concerns extensions to READ and WRITE. The first extension is an option that permits the READ or WRITE to receive or send only a single character. The single character READ is indicated by prefixing the variable name specified in the READ argument with an asterisk, as in

```
READ *XYZ
```

This command specifies that the first character received from the current device will be interpreted according to a set of rules defined by the implementor, and an integer value corresponding to the character entered will be returned. Conversely, the command

```
WRITE *XYZ
```

causes the integer value of XYZ to be converted to a single character according to the implementation's defined rules, and this value is then transmitted to the current device. This extended command is sometimes useful in performing special functions on output devices, or in implementing special communications protocols.

In order to prevent a READ operation from waiting indefinitely for a response from some device, it is possible to place a time limit on the READ command by using the timeout extension described in the OPEN command. The time limit is specified in seconds, and appended to the read argument. Thus, the command line

```
READ !,"WHAT IS YOUR ID?",ID:10
```

gives the repondent ten seconds to enter the appropriate identification number. At the end of that time, only those characters already entered will be returned by the MUMPS system. The special variable $T will be set to reflect the result of the timeout. In the case of a single character READ as described above, the result will be an integer specified by the implementation. If no single character is received, the value -1 is returned.

$JUSTIFY Function

In chapter 6 we showed how the format control character "?" can be used to create columns of data aligned at a particular character position. Alignment of items at the start of the string is called "left justification". It is sometimes convenient or necessary to align a set of items at the right end of the string, or to "right justify" the text. Right justification is often used in tables of numbers or monetary values. Consider the following example used to right justify the variable X at Column 20:

```
FOR I=0:1:20-$LENGTH(X) WRITE " "  ;PAD WITH SPACES
WRITE X
```

Since this method requires quite a bit of computation that might prove time-consuming in printing a long list, MUMPS provides a function, $JUSTIFY, to do the padding for us. $JUSTIFY(X,L) pads the string expression X on the left with spaces to a total length of L. If X is already longer than L, X itself is returned, i.e., no information is lost. Using $JUSTIFY, we can rewrite our example as follows:

```
WRITE $JUSTIFY(X,20)
```

Note that a convenient way to create a string of ten spaces would be

```
SET X=$JUSTIFY("",10)
```

When used with an optional third argument, $JUSTIFY formats numeric quantities. $JUSTIFY(X,10,2) takes the numeric interpretation of X, rounds it to two decimal places, and pads the result to ten characters. Remember that integer division ("\") can be used to truncate numeric quantities. $JUSTIFY provides the complementary rouding operation. The third argument of $JUSTIFY must be non-negative (a value of zero specifies that the number will be rounded to an even integer). Negative values are currently illegal; however, the designers of Standard MUMPS left open the possibility that the function will at some future date be expanded by specifying actions taken with negative values.

The following examples illustrate some uses of $JUSTIFY.

Expression	Value Produced
$JUSTIFY(3.716,0,0)	4 (note no decimal point)
$JUSTIFY(3.716,0,1)	3.7
$JUSTIFY(3.716,4,2)	3.72
$JUSTIFY(3.716,0,5)	3.71600
$JUSTIFY(7,0,3)	7.000
$JUSTIFY("ABC",0,0)	0

Summary

This chapter has introduced some extensions to the READ and WRITE commands that enable the user to access devices other than his terminal. These extensions are essential in the preparation of reports requiring high speed line printers, and they are often necessary when a large file of data are to be entered into the MUMPS system. The necessity of adding implementation specific constructs to the OPEN, USE and CLOSE commands facilitating these actions is an important feature of the language that should be checked before these commands are used.

Chapter 12

Global Variables

In Chapter 2 we introduced the concept of a variable, an entity that holds a value. This concept was expanded in Chapter 10 to include the notion of arrays. Arrays provide a convenient method of grouping associated values in clusters with common relationships. Now we want to introduce the principle of variables that will be retained by the MUMPS system even after the user has signed off.

The variables described up to this point in the PRIMER belong to the the class called "local variables". MUMPS uses the term "local" in a somewhat special sense, in that local variables are limited both in their accessibility and their lifetime. MUMPS local variables may only be referenced by a single user during a single session; they cease to exist when he signs off, and they may not be accessed by other users even though they are logged in at the same time. This notion of the meaning of "local" conflicts with the use of the term in some other programming languages. Users familiar with other systems such as APL and ALGOL or PASCAL should take note of the distinctions and use the term with due caution.

Since one of the main benefits of the MUMPS language relates to its use of shared files, some means must be found to define variables that may be shared by many users and that will remain valid independent of the user who may have created or modified them. MUMPS accommodates these requirements through the use of <u>global variables</u>. Global variables have the same general properties as local variables, but they may be shared by multiple users, and they are stored by the MUMPS system until they are explicitly KILLed. (Some implementors define additional protection mechanisms which limit the way in which global variables may be accessed or modified, thereby providing an extra level of protection for purposes of confidentiality and security). The most common way in which global files are retained is on some mass storage device, such as a magnetic disk. The physical characteristics of such devices may affect the manner in which they are accessed, but from a logical point of view their nature remains constant. As you become more familiar with your system and your own needs for shared global files, you may wish to learn more about the implementation features of your system's global files, so that you can make efficient use of these files in your own programs.

Global variables are distinguished by the presence of a caret or up-arrow (^) in front of the variable name. In all other respects, global variables follow the same rules of formation defined in Chapter 10 for local variables: they may be subscripted or non-subscripted; if they are subscripted, they may have an unlimited number of subscripts (providing the line length is not exceeded). Generally speaking, global variables may be used intechangeably with local variables. They return similar results as local variables when used with $DATA and $NEXT, and they obey the same rules when referenced in SET and KILL commands.

There are, however, three situations in which global variables may not be used:

1. as the control variable in a FOR command;
2. as the argument of a READ command; and
3. as a subargument in the parenthesized form of the KILL command.

Naked References to Global Variables

An additional, shorthand, form of global variable reference is also provided in the MUMPS language. This form, referred to as the naked syntax, uses a partial subscript list whose complete definition depends on the last subscript used in a previous global reference. For example, the following SET statement:

```
SET ^A(1,2,3)=3,^A(1,2,4)=4,^A(1,2,5)=5
```

May also be written, using the naked syntax, as

```
SET ^A(1,2,3)=3,^(4)=4,^(5)=5
```

In this example, we make use of all but the last subscript in the first SET command in order to complete the definition of the remaining two assignments. This "full" reference, then, defines an imaginary string called the "naked state," which consists of the global name, the left parenthesis and the subscripts up to but not including the last. The effect of a "naked" reference is to append the subscripts that follow the caret and left parenthesis (used immediately following the caret to denote a naked refernce) to the "naked state" subscripts and then execute the resulting fully defined

reference. Thus, ^(4) in the previous example is expanded by the MUMPS interpreter to the full term ^A(1,2,4).

Naked references may also use multiple subscripts, so that the command line

```
SET ^A(1,2)=5,^(2,3)=13
```

results in ^A(1,2,3) being set equal to 13.

Naked references are used to keep code concise, to avoid the use of indirection (see Chapter 14) by permitting "relative" addressing of global data, and, in some implementations, to achieve increased efficiency in the manipulation of global variables. This form of reference may, however, be a bit tricky, so it should be used with caution. Remember that the naked state is defined by the last executed global reference, not necessarily the one that precedes it in the routine.

As an example, consider the following sequence of instructions. Assume that we wish to test for the existence of ^A(I) and, if defined, create ^A(I,J)=J for values of J from 1 to 100. The following command line appears to be a reasonable solution:

```
IF $DATA(A(I))  FOR J=1:1:100 SET ^(I,J)=J
```

However, when this sequence of instructions is executed, we will create a global file with 101 levels, since each execution of the naked syntax will increase the length of the naked state by one subscript. The resulting gobal will be:

```
^A(I,1)=1

^A(I,I,2)=2

A^(I,I,I,3)=3
```

etc. In fact, the correct code to accomplish the desired task is

```
IF $DATA(^A(I)) SET ^(I,1)=1  FOR J=2:1:100  SET ^(J)=J
```

To consider another example, the line

```
SET ^A(1)=1,^(2)=^(3,4)
```

sets ^A(3,2) equal to the value of ^A(3,4), rather than setting ^A(2) equal to ^A(3,4). Remember that the righthand side of a set is evaluated (resulting in a global reference) before the lefthand side is assigned.

The use of the naked syntax is erroneous if the naked state is not defined. The naked state is undefined in the following cases:

1. At user sign-on, i.e., before any global references have been made

2. After any unsubscripted global reference.

3. After any $DATA or $NEXT of a subscripted global reference when the final level of the specified reference does not exist (i.e., when the $DATA returns a value of zero). For example,

```
$DATA(^A(1,2,3,4))
```

sets the naked state to an undefined value if $DATA (^A(1,2,3))=0.

LOCK Command

In any multi-user environment, different tasks must be able to synchronize their activities. Synchronization is necessary to prevent errors and inconsitencies in shared data bases. Consider a set of tasks appending transactions to a list. The list is kept in global array ^L. In order to keep track of the manner in which the list is growing, ^L(0) is used to store the index of the most recent entry, i.e., if ^1(0) is 10, the latest entry is ^L(10). Suppose each user adds to the list using the folowing command line:

```
SET INX=^L(0)+1, ^L(0)=INX, ^L(INX)=DATA
```

Without synchronization, it is possible for one user to be interrupted after the first SET argument, and a second user might subsequently execute the whole line of code. Both users will have used the same index with the result that the second transaction

(actually the first one stored) will be lost. This type of problem occurs whenever a transaction must change a number of related nodes, or records, in a data base. If any of the nodes contain logical pointers, as in the example given above, an inconsistency or omission in the data base may occur.

This type of problem can be avoided by making sure that only one user can be executing the critical sequence of code at one time. The LOCK command provides a method whereby a user can delimit this code sequence, and prevent other users from executing similar critical code at the same time.

The LOCK command requests exclusive ownership of a logical resource. There is no association between the name and any real variable or data base. LOCK ABC merely requests exclusive ownership of the name ABC. If any other user executes a LOCK ABC command, he will be temporarily suspended until the original ownership is released. A LOCK command without arguments releases all currently owned resources.

Each time the LOCK command is executed, the MUMPS system will first release all resources currently owned by that user. If an argument is present, ownership of that name is subsequently attempted. This sequence of actions avoids the possibility that a user might in fact be waiting on ownership of a resource he has already obtained (the deadly embrace or deadlock situation).

To resolve the problem raised in our example, we bracket the critical code sequence (herein referred to as the critical region) with LOCK commands. If we call our resource LIST, then

```
LOCK LIST SET INX=^L(0)+1, ^L(0)=INX, ^L(INX)=DATA LOCK
```

assures the synchronization of insertions. All routines which perform insertions must, of course, bracket their critical regions in the same manner.

LOCK ABC will suspend execution until ABC can be owned. Since this situation could continue indefinitely, excessive delays can be avoided by appending a timeout to the argument, for example LOCK ABC:5. This timeout, similar to the ones described in Chapter 11, will result in suspension of execution for no more than 5 seconds. If ownership cannot be obtained within that time period, $TEST is set

to 0 (false) and execution continues. If ownership is obtained, $TEST is set to 1 (true) when ownership is obtained ($TEST is not affected if no timeout is specified).

Note that LOCK A,B will not obtain ownership of both A and B. Since, as noted above, every LOCK first releases previous ownerships, and since LOCK A,B is equivalent to LOCK A LOCK B, only B will be LOCKed. When ownership of more than one resource is desired, LOCK (A,B) should be used. In this multiple form, ownership of all names in the list must be obtained. MUMPS will not, for example, obtain ownership of A, then wait for B to become avaialble. In such a situation, unless A and B are both available, neither will be obtained.

It is sometimes useful to have different levels of interlocks in order to avoid bottlenecks. In a patient record system, for example, changes to a patient's file must be protected. If the resource name is the same for all patients, only one patient's file could be updated at a given time. If the resource name is keyed to the individual patient, many different updates could be accomplished simultaneously, thus increasing system throughput. On the other hand, some functions, such as entering a new patient to the file of changing a system-wide index may affect data common to all patients. These operations must have critical regions which preclude all patient updates as well as other system-wide updates. To accomplish this hierarchical protection, an array form of LOCK can be used. If we write

```
LOCK ^PAT(ID1)
```

we control the critical region for updating the record of the patient with identification number ID1. Another user might update a different patient's record by executing

```
LOCK ^PAT(ID2)
```

In order to obtain ownership of the entire resource,

```
LOCK ^PAT
```

would be executed, requiring that ownership of all lower level resources be available. Once ^PAT is onwned, system wide updates can be performed, and ^PAT(ID) ownership cannot be obtained since it lies in the subtree of ^PAT. This hierarchical aspect of the MUMPS LOCK command permits clear and concise implementation of complex synchronization requirements.

In the example given above, we have used the LOCK command with

respect to a global variable, showing how it may be partially or completely protected from dual updating. Remember, however, that LOCK does not refer to any physical structure and does not, by itself, imply any data file protection. The synchronization of critical regions is done purely by programmer convention. System management policies must be relied on to ensure reliability of this command in a given implementation.

Summary

In this chapter we have introduced the concept of global variables, values which may be shared by multiple users of a MUMPS system and which may be retained by the MUMPS system for extended time periods. Local variables, on the other hand, are used during individual interactive sessions, may not be shared by users, and are not retained after the user who set them signs off the system.

One of the principal strengths of the MUMPS language is its ability to create global variables. In many respects, these shared variables bear similarities to the database systems defined in some other languages. Two of the principal differences are that these variables need not be declared before they are used, and they may be sparse arrays, as defined in Chapter 10. These distinctions serve to enhance the flexibility and versatility of the language. At the same time, user control of the creation and modification of global variables opens the door for abuse of such a privilege, and safeguards must be exercised, either at the system of programmer level, to maintain the validity of a shared data base.

Chapter 13

Some Special Features of Standard MUMPS

We have considered nearly all the standard features of the MUMPS language up to this point. There remain, however, a few language elements that do not readily lend themselves to inclusion in previous chapters. This chapter will deal with those features - Pattern Match, $HOROLOG and $TEXT - language characteristics that provide additional tools helpful in editing and running MUMPS routines.

Pattern Match

MUMPS has one further string relational operator, the Pattern Match operator. This feature has not been introduced up to this point because it is somewhat more complicated than the operators described in earlier chapters, and because many applications do not require the pattern match capability.

In some situations, it is valuable to test whether an input string such as a patient name, inventory number, date or some other coded entry is entered according to system specifications. If, for example, we wish to make sure that date is entered "MM/DD/YY" with the slashes included, we would like to find a convenient method for testing the validity of the data entry. The pattern match operator provides us with a direct evaluation of the string that obviates the necessity of a character-by-character comparison of every position in the input string. The test of the data string is made by evaluating it against a special data element called a pattern. The expression

```
VAR?pattern
```

is true if the value of VAR "matches", or is in the form of, the pattern. Likewise,

```
VAR'?pattern
```

is true if the value of VAR does not match the pattern.

To understand this concept, we must explain the variations possible within the definition of a pattern. A pattern is a sequence of pattern elements, each of which is composed of a pattern code or string literal preceded by a multiplier count. There are seven basic codes that may be defined within a pattern. They are

A - the 26 upper and 26 lower case alphabetics

C - the 33 ASCII control characters (including DEL)

E - the entire ASCII character set

L - the 26 lower case alphabetics

N - the 10 numerics

P - the 33 punctuation characters

U - the 26 upper case alphabetics

In addition, specific characters or strings may be identified by enclosing them in quotation marks.

Using these definitions, we might verify a date as being in the desired MM/DD/YY format by describing this pattern as

2N1"/"2N1"/"2N

In this description, the numeric elements 2N are separated by slashes, and only one occurrence of each slash is permitted between each number pair. Another example might be an inventory part number that must appear in the form "12-345". A pattern to verify this format would be

2N1"-"3N	pattern
2N 1"-" 2N	pattern elements
"12" "-" "345"	string segments

Individual pattern codes relate to the ASCII character set, as noted above. To increase the versatility of this operator, composite pattern codes may be specified. Thus

AP - selects either alphabetics or punctuation marks (as does PA)

LN - slects lower case and numeric characters (as does NL)

The multiplier may be explicit, an integer specifying the exact number of characters within the specified set, or indefinite, in which case the period "." is used. If the indefinite multiplier is used, any number of matching characters (including none at all) will satisfy the specification. The indefinite multiplier permits the pattern match operation to be adaptive, searching for an appropriate decomposition of the input string.

Consider the following examples:

specification	meaning
X?1"ME".E	X begins with "ME" and is otherwise unspecified
X?.E1"ME".E	X contains "ME"(same as X["ME")
X?.E1"ME"	X ends with "ME"

To test for a patient name in a last name, comma, first name format, the specification

PAT?1A.A1","1A.A

will require that there be at least one alphabetic character in both last and first names, that they be of indefinite length but composed only of upper and lower case alphabetic characters, and that they

must be separated by a comma. If we wanted to permit (but not require) spaces after the comma and before the first name, we could use the pattern

```
PAT?1A.A1","." "1A.A
```

which covers all valid possibilities. To illustrate with another example, if we wanted to test for a valid non-subscripted variable name (using the definition provided in chapter 2), we could specify

```
NAM?1"%".NA!(NAM?.1A.NA)
```

which would correctly identify a variable beginning with either an alphabetic character or a percent sign and followed by alphanumeric characters.

$HOROLOG Special Variable

In many computer applications, it is valuable to have access to the date or even the time of day. Bills, reports, transactions, and many other documents are routinely dated. Some transactions that may take place several times during a day need in addition to have the time of day recorded. Programmers sometimes want to know how long it takes for a certain section of code or activity to be completed, in order to evaluate the performance of the system or the code itself. Many computers can accommodate this type of requirement by providing an "external clock", a device that keeps track of the time of day, often to the nearest fraction of a second. For systems in which an external clock is available, MUMPS offers the $HOROLOG function to permit the user to obtain date and time information in his programs.

$HOROLOG returns a string which contains two numbers separated by a comma. The form is "D,S" where D is the day (using a formula described below) and S is the number of seconds elapsed in that day at the time the function is called. In order to allow the first part, D, to be completely generalizable for any day, month or year, $HOROLOG provides this day as a single number that represents the number of days elapsed since the first day of January, 1841 (the designers presumed that most MUMPS applications could operate within the period 1840 to the present without difficulty). Thus, Christmas Day, 1977 has a D value of 50034. You may wish to check this by remembering that every four years (including one leap year) has a total of 1461 days, and that the year 1900 has an extra day.

It is not particularly convenient for us to remember this type of calculation, so we quote below a MUMPS routine that can be used to calculate the date from the $HOROLOG D value given by your MUMPS system. This routine is taken from the MUMPS Programmers' Reference Manual, p. 98.

```
SET H=$HOROLOG>21973+$HOROLOG,LEAPYRS=H\1461,Y=H#1461

SET YEAR=LEAPYRS*4+1841+(Y\365),DAY=Y#365

SET MO=1 IF Y=1460 SET DAY=366,YEAR=YEAR-1

FOR I=31,YEAR#4=0+28,31,30,31,30,31,31,30,31,30,31 QUIT:DAY'>I
     SET DAY=DAY-I,MO=MO+1

WRITE MO\10,MO#10,"/",DAY\10.DAY#10,"/",YEAR
```

You may want to study this routine a little to see how it works. The first line adds one to the days if the date is past 1900, then determines how many complete leapyear cycles there have been and how many days remain after those cycles have been accounted for. The next line adds the base date of 1841 to the date in years, and calculates the number of days in an incomplete year remaining at the end. The third line adds the appropriate number of days for each month, including leap year, until the days calculated in the previous line is met, and also calculates the month. The last line, using modulo and integer divides, writes out the date in an "MM/DD/YY" format with a leading zero where a month or day is less than 10. Studying a sample routine of this sort may be quite helpful in gaining an understanding of several programming techniques useful in the MUMPS language.

The Seconds portion of the $HOROLOG function may be extracted from the first part of the string by using $PIECE:

```
SET TIME=$PIECE($HOROLOG,",",2)
```

the hours can then be obtained by integer division of TIME by 360, the minutes by the formula TIME/60#60, and the seconds by TIME#60. You might want to try writing the result so that, by using $PIECE, you add either AM or PM to the time as appropriate (if you need help, see the Reference Manual, page 98).

$TEXT

One of the requirements of an effective computing environment is the ability to edit programs. In some languages, the operating system is left the responsibility of performing all editing functions. Often, this approach leads to wide diversity in the ways in which system editors are implemented. MUMPS, having been developed in part to incorporate features of an operating system with those of an interpreter and file management system, has taken a slightly different approach. Some of the functions normally reserved for operating systems are incorporated in the interpreter itself, though, as we will see in Chapter 15, a number of these features depend for their specific interpretation on the implementor's judgement.

$TEXT is an example of a function that is particularly useful in editing routines. The purpose of $TEXT is to return to the user a command line, as specified in the argument of the function. As we have seen in Chapter 7, a command line may optionally begin with a label. A routine will frequently have several lines with labels interspersed with other lines containing only commands. The identifying argument of $TEXT permits identification of either a labelled or an unlabelled line by providing an offset from a line having a label. For example, let us suppose that we have a routine as shown below:

```
ADDRESS ;CREATED 11/78 BY FNU
     ;ACCEPTS ADDRESS INFORMATION. QUITS ON NULL ENTRY
     WRITE !,"PLEASE ENTER NAME, STREET ADDRESS, CITY, ZIP AND STATE"
     WRITE !,"AS PROMPTED. TO QUIT, ENTER BLANK NAME."
START FOR I=1:1 QUIT:^NAM(I)=""
     READ !,"NAM: ",N IF N="" GOTO FINISH
     SET ^NAM(I,1)=N
     READ !,"STR: ",N SET ^NAM(I,2)=N
     READ !,"CIT: ",N SET ^NAM(I,3)=N
     READ !,"ST: ",N SET ^NAM(I,4)=N
     READ !,"ZIP: ",N SET ^NAM(I,5)=N, I=I+1 GOTO START+1
FINISH WRITE !,I," ENTRIES NOW IN GLOBAL FILE NAM." QUIT
```

Supposing that we wished to make reference to several of these lines. By using the $TEXT function, we can access any line in this routine through the appropriate use of the $TEXT arguments. Examples of the use of $TEXT are given below:

Expression	Value Produced
$TEXT(START)	START FOR I=1:1 QUIT:^NAM(I)=""
$TEXT(START+4)	READ !,"CIT:",N SET ^NAM(I,3)=N
$TEXT(+9)	READ !,"CIT: ",N SET ^NAM(1,3)=N
$TEXT(FINISH+5)	(null string)

Note that $TEXT can use either an absolute integer offset from the start of the routine (the first line is line 1) or it can use a label with an optional offset in order to identify a particular line.

Given the ability to identify any command line in a routine, it is a far simpler matter to write editing routines that will identify and replace specific components of that line. Several other functions and commands such as $FIND will be helpful in this regard.

Summary

In this chapter, we have covered several language elements useful for programming and reporting that do not lend themselves to inclusion in previous chapters. First, we chapter described the pattern match string operator, useful in checking the format of input strings without recourse to complex code identifying the specific fields to be checked. Though the applications of the pattern match operator are limited, they serve a very valuable function in certain situations. With the description of this operator, we have covered all operators provided in the MUMPS language. Next, we discussed the $HOROLOG speical variable used to obtain date and time information for report writing and program evaluation. Finally, we ilustrated use of the editing function, $TEXT. The next two chapters will complete description of all language elements in the MUMPS language.

Chapter 14

Indirection

Indirection is a facility that provides the MUMPS programmer the ability to dynamically create and execute MUMPS code. Within certain restrictions, the programmer can cause the value of an expression to be, in effect, inserted temporarily into the program, thereby altering the execution steps performed.

In MUMPS, indirection may occur at four levels: Commands, Arguments, Names and Patterns. Of these four, the simplest is the Command level indirection. Indirection at the Command level is accomplished by using the XECUTE Command. Each argument of the XECUTE Command is evaluated as an expression. The resulting expression is then interpreted as if it were a Command Line, and the instructions are executed. For example, if we wrote the following statement:

```
SET L="WRITE !,""I AM XECUTED"""
```

and we later wrote the statement

```
XECUTE L
```

the phrase "I AM XECUTED" will appear in quotes at the start of a new line. The value of the variable, L, was treated as a Command Line and executed as if it had been inserted into the program at the position of the XECUTE L Command. If a Command such as XECUTE L is placed in a loop where it is executed repeatedly, the value of the variable L can be reset prior to each execution, and the output modified accordingly.

The XECUTE command may have multiple arguments, evaluated and executed in sequence. Consider, for instance, the Command Line

```
XECUTE L,"S Y=3.7)","DO ^ABC"
```

Each of these arguments will be executed in left-to-right order, concluding with the full execution of routine ABC.

Arguments to XECUTE may also be conditionalized by appending a postconditional to the individual argument, as discussed in Chapter 8. A situation where this technique might be useful is the following. Consider assigning a debugging Command Line to the variable D, with the intent that the debug statements be executed only when some condition DB is set true. In such a case, the Command Line

```
XECUTE A,D:DB,B
```

would always result in execution of A and B, but would execute D only if DB evaluates as true.

The next three forms of indirection (argument, name and pattern) require a special syntax. In these contexts, indirection is specified by an indirection operator "@" followed by an indirection expression. The formal definition of the language specifies that the expression used in indirection must be an expression atom (expratom), rather than any form of expression. In simple terms, this restriction means that if the expression selected is not a variable name, a constant, or a function, it must be enclosed in parentheses. Hence, A, Q(3), ^B, "S X=3", (A-B-C) and $PIECE(S,D,M) are all valid expressions to be referenced in indirection contexts, whereas A-B-C is not valid, since it is not enclosed in parentheses.

Argument level indirection provides the ability to use the value of an expression atom to specify one or more complete arguments of a Command. This type of indirection is indicated by the presence of the indirection operator @ at the first position of a command argument. When this operator is detected, the expression atom following the @ is evaluated then executed as it were one or more complete arguments to the command. To illustrate, assume that the following variable assignments have been made:

```
X1="A=3"  X2="B=6,Y=Z"
```

The command SET @X1,@X2,Q="ABC" would result in assigning A the value of 3, B being set equal to 6, Y to the current value of Z, and Q tothe string ABC. However, the Command SET @X1+Z would be illegal since the indirection must constitute the complete argument specified in the SET Command.

Argument level indirection is permitted for all commands which

accept arguments, with the exception of the FOR Command, where no Argument level indirection is allowed.

Name level indirection is only permitted in a few restricted situations in the MUMPS language. As in the case of Argument level indirection, Name level indirection is identified by an initial @ operator followed by an expression atom. The expression atom is evaluated and used in the position occupied by the indirection itself. The value of the expression atom must be a legal MUMPS name. The places in which this level of indirection is permitted are the following:

1) Whenever a variable name (either local or global) is permitted. Thus, if X=A, the Command SET @X=4 would result in setting A equal to 4.

2) Wherever a line label is called in a Command or Function, as in the statement GOTO @X^ROU, where X="LINE1", a label in the routine ROU.

3) Wherever a Routine name is referenced by a Command, such as DO LAB^@X, where X="^RTN", the name of some routine;

4) to provide name references to the LOCK Command, as in L @X, where X="^A(1,2)".

Pattern level indirection is very similar to name level indirection. Its use is quite specific, however: the indirection operator must immediately follow the pattern match operator "?", and the indirection argument (expression atom) must evaluate to a valid MUMPS pattern. In the following examples, assume that A="1A", B="1N.N1A", and C="CA".

Program Code	Executed Code
X?@A	X?1A
X?@(A-B)	X?1A1N.N1A
X?@C	X?1A

The last example illustrates that indirection can be nested, such that one indirect expression may be evaluated and found to contain yet another level of indirection. This ability to nest indirection is not restricted to the Pattern level, but may also

occur within Command, Argument and Name indirections.

Summary

Indirection is a powerful tool, and as such it may provide the user with considerable flexibility in programming. For many reasons, however, indirection should be used with moderation and care. Code written depending on indirection is harder to analyze and document, since the self-modifying aspects may be difficult to trace. Safe programming practices dictate reserving its use to those situations in which it is difficult to accomplish the task withouit indirection, and then to document with liberal comments the intent and range of indirection envisaged.

Chapter 15

Implementation - Specific Components of MUMPS

The ANSI Standard for MUMPS includes a set of commands, functions and variables that are highly implementation specific. It is the purpose of this chapter to discuss these topics briefly, so that you will be aware of their existence. For detailed information on their use in your environment, consult the manufacturer's documentation or Chapter 18, if it is included in your edition of this PRIMER.

BREAK and VIEW Commands, $VIEW Function

The BREAK Command was defined as part of Standard MUMPS to provide an access point for nonstandard programming aids. When used without an argument, execution of the BREAK command causes the program to be suspended until receipt of some signal (unspecified in the standard, therefore implementation specific). uhus, it "freezes" execution of the program at some point selected by the programmer, allowing him to use other features of the machine to examine conditions, status of variables and so on, then to resume execution. The standard also permits implementation of specific arguments to be defined and provided by each implementor.

VIEW is another standard Command with implementation-specific arguments. The standard states that VIEW is provided to give the implementor a means of examining implementation-specific information. For example, the implementor might use VIEW to permit the programmer to "dump" some portion of memory in order to examine data or code as a debugging aid.

The $VIEW function is similar to the VIEW command in that it is also used to examine machine dependent information. The argument is unspecified. Generally, this function is used to return one specific piece of machine dependent information for each argument.

Use of the VIEW command and $VIEW function renders a program non-portable, since its operation will vary from one system to

another. For this reason, programmers interested in sharing their code should restrict use of these commands.

$JOB and $STORAGE Special Variables

The $JOB and $STORAGE special variables are somewhat specific to individual implementations. The first, $JOB, is a special variable which returns a positive integer that uniquely identifies the MUMPS process. In other words, it is a job number that is assigned to you when you begin a MUMPS session. It remains constant throughout that session, and uniquely identifies you so that your activity may be separated from other users. It is often used as an initial script in global scratch files to separate individual users' areas.

The $STORAGE special variable is perhaps more generally applicable, particularly for smaller MUMPS systems. At each reference, $STORAGE returns an integer which is the number of characters of available space remaining for a given user. The exact interpretation of this number, in terms of the number of command lines or other measures, depends on your particular implementation.

Implementation Defined Commands, Functions and Special Variables

As you learned in earlier chapters, MUMPS permits all commands, functions and special variables to be abbreviated by using only the first letter of the name. Thus, SET may be written S, $FIND as $F, etc. The Standard did not define commands, functions and variable for every letter of the alphabet, but it did reserve all letters as possible commands in the future. This reservation places an inherent limitation on MUMPS systems. Users wishing to implement their own instructions are prevented from arbitrarily selecting an unused letter in view of this restriction. To make it possible for an implementor to introduce new commands, functions or special variables, a convention has been adopted that permits extension of commands through the single initial letter Z. The Z indicates only that a non-standard command, function, or special variable follows this introductory letter. The rest of the name may then be used to indicate whatever name is appropriate for the action intended. As an example, if an implementor wished to provide a square root function, he might wish to name it $ROOT. Since $R is already reserved for $RANDOM, he would instead name his function $ZROOT, which would be abbreviated $ZR. The initial Z identifies this command as implementation specific. The same practice may be used in defining commands and special variables. Note that, in these cases, the abbreviated form must include the Z plus the next letter in order to

distinguish it from other Z-instructions. Most implementations make use of this extension, so you should check carefully any implementor-supplied documentation to learn of those features that have been added to your system. Remember too, however, that use of these extended features makes your Routines non-portable.

Direct Mode

Up to now, we have been concerned only with the execution of MUMPS commands as part of a stored MUMPS routine. This form of execution is usually referred to as "indirect" or "programmed" mode of operation. Many implementations of MUMPS also permit "direct" execution, a mode in which commands are executed as they are typed in at the terminal. In such an implementation, entering the command

```
WRITE 1+2+4/2
```

would result in

```
3.5
```

appearing on the terminal immediately after the line was typed in. In other words, the command would be executed "directly", and not stored for future use. Such direct mode operation is frequently used to enter, modify and save routines, as well as to initiate execution of saved routines. For example, if a routine named ADMIT has been stored on the MUMPS system, one common technique for invoking this Routine is to require the user to type

```
DO ^ADMIT
```

in direct mode. In this case, the user is requesting a transition from direct mode to indirect execution of stored code. When the ADMIT Routine has completed execution of its steps, the system will once again cause transition from indirect mode back to direct mode, allowing the user to request some further action.

Another valuable use of direct mode is to debug program steps. Frequently, the BREAK Command defined in the previous chapter is programmed in such a manner that its execution causes the current Routine to be suspended and the user to be returned to direct mode. In this environment, the user may examine the state of his Routine,

perhaps change some conditions, then return to indirect execution of the stored Routine.

Any implementation having direct mode will have some restrictions as to the operations that may be performed only on either direct or indirect mode. These decisions are not standard, and you should refer to your system documentation for details.

Summary

We have now treated all MUMPS language elements defined by the ANSI standard, and we have also introduced the method whereby individual implementors may add new features to the language. In the remaining chapters, we will examine the directions that MUMPS may take in its evolution as a computing language, and we will illustrate its use with some specific examples.

Chapter 16

The Future of MUMPS

This PRIMER has attempted to describe the MUMPS language as it is defined in the ANSI Standard. MUMPS is now in widespread use around the nation and the world. There are over 1200 installations using MUMPS, and a growing number of manufacturers have announced their support of MUMPS as a language available on one or more of their systems.

Computer languages are not static. Widespread use leads to innovative suggestions as to language modifications and improvements. The original history of MUMPS (traced briefly in Chapter 1) documents the initial spread and fragmentation of the language into many dialects - a fate that has befallen many computer languages. The creation of the new Standard MUMPS is a tribute to the foresight of the federal sponsors who gave MUMPS implementors support as they wrestled with the problems of its definition. Once accepted, it is reasonable to assume that evolutionary trends will once more exert pressure to modify the language in accordance with the perceived demands of its users.

What sorts of changes might we expect from this pressure? Will the language once again fragment into incompatible dialects? Are there any controls to is growth that offer hope for continuation of the Standard? Conversely, is there any hope for growth of the language as demand for certain features becomes widespread? The answers to these questions are encouraging. There is hope for controlled change. That process of language evolution has already begun, and it is possible to make some general statements regarding the future evolution of MUMPS based on the experience gained thus far. This chapter deals with these questions, leaving open the ultimate question of the acceptability of the language as a semipermanent member of the computing tools widely used in business, health care, and academic circles.

The Process of Change

When MUMPS was adopted as a Standard language by the American National Standards Institute (ANSI), it entered into a controlled environment as a price for acceptance. ANSI regulations require that

the standard remain in effect for five years following its acceptance, at the end of which time it must be reviewed and either renewed as before or revised in accordance with certain procedural regulations. The MUMPS Development Committee has been named the official body responsible for monitoring the review of MUMPS. Membership in the committee is open to interested groups who accept its constitution and bylaws. Proposed revisions to the Standard must be submitted prior to September, 1982 (the end of the first five year period), when they will be considered by ANSI for acceptance in the revised language definition.

The role of the MUMPS Development Committee in this process is therefore to consider possible additions, corrections or modifications to the language and its supporting document. Already, a number of minor inaccuracies in the original language definition have been detected by its users, and the corrections duly noted in revised documentation. In addition, the Committee has considered possible language extensions, knowing full well that any potential change must be thoroughly evaluated prior to its adoption. No formal language change can be made to the Standard before 1982. Individual implementors are free, however, to test extensions and report on their effect on the language. This process will eventually lead to the informed adoption of certain language modifications in 1982.

The By-laws of the Development Committee indicate clearly how a proposed language modification may be accomplished. Assignment of a proposed change to the appropriate subcommittee is the first step. Favorable recommendation for the change is then considered by the full Development Committee, who will add the proposed revision to the document to be submitted to ANSI in 1982. In view of the responsibility that this process places on the MUMPS Development Committee, it is logical to assume that evolution will be gradual. Nonetheless, there are some features that do appear likely to gain acceptance in the next revision of MUMPS. Some of these features are briefly discussed below.

New Language Features

Of all the proposed language changes that have been proposed to date, one appears virtually certain to become a part of the Standard at the next revision date. That feature is the capability to use string subscripts. A string subscript may be defined as a subscript whose values include characters from the ASCII printable subset. Examples might be

```
INVNTORY("ITEM1"), ^PATIENT("SMITH"), LABTEST(DATE,TIME,"HCT"), etc.
```

This feature represents a significant extension over the integer subscripts now permitted in the language. With the sparse array concept and automatic ordering already contained in the language, the alert programmer will note that sorting (a major requirement in most string handling operations) becomes implicit in many programmatic situations. Because of its widespread appeal, this feature has already been tested with widespread acceptance by several implementors. Based on this favorable reception, the MUMPS Development Committee is on record as approving this proposed modification to the language for inclusion in the 1982 revision.

The foregoing example illustrates the process by which changes may be adopted in the language. Other changes that are currently under consideration relate to the desirability of identifying special arguments to be "passed" to functions or subroutines, the possible need for definition of Direct Mode actions, revisions of certain supporting documentation, and other features which are as yet insufficiently articulated to describe.

Limitations to Language Evolution

Several trends have already become apparent in the deliberations of the MUMPS Development Committee since the Standard was adopted. First, it is clear that no language modification will be accepted without careful scrutiny. There is always a faction proposing rapid change which opposes a faction preferring little or no change; in the case of MUMPS, the balance appears to be in the direction of very gradual change without a clear-cut obligation to add new features as soon as they are proposed. This condition is believed healthy for the language, permitting implementors to minimize their language maintenance problems while at the same time allowing for gradual evolution in the direction of broadly accepted new features.

Second, it appears highly unlikely that the language will fall into the trap of attempting to be all things to all people. Specifically, the weakness of MUMPS in the area of scientific calculation is one that is likely to remain. Because of its design, it will never be able to compete successfully with languages such as APL and FORTRAN designed to facilitate computation. Its users are not willing, therefore, to burden the language with features permitting complex number arithmetic, vector and matrix operations and special trigonometric functions to enhance its strength in this area. Such features can be written and shared as application routines, but their contribution to the language will probably remain secondary to the string and file manipulating features that characterize the current version of the Standard.

Portability as an Aid to Evolution

We have noted previously that an attractive feature of MUMPS is its portability between standard implementation sites. We also pointed out that individual implementors may, by use of the "Z" escape feature of the language, implement certain extensions to the language. It appears likely, therefore, that the increased exchange between different users will lead to greater uniformity in those language extensions that have been found particularly useful in two or more institutions.

Exchange of application packages may lead to the adoption of I/O conventions, for example, that may become acceptable as standard features. Use of specific editing packages may also result in acceptance of some conventions facilitating documentation or program modification. As more experience is gained through the experience of the MUMPS Application Library, it appears likely that a consensus for certain changes may arise, leading to formal adoption of the proposed changes through the process outlined above.

Effects of Technological Advances on the Evolution of MUMPS

One of the more dangerous realms for speculation in today's world of rapidly advancing technology is to speculate on its effect on existing institutions - in this case the MUMPS language. However, a few guarded observations may be permitted to place in perspective those changes that have already become accepted in today's marketplace.

MUMPS was originally implemented on minicomputers. It has since been implemented on several large mainframe systems, and more recently it has been implemented on microprocessors. As more experience is gained in the continuum of computing hardware, it appears likely that some changes in the language may result.

The separable features of MUMPS include its function as an interpreter, as a file manipulation system, and as an operating system. The modularity of microprocessors makes it likely that the MUMPS interpreter will be separated from the global file handler. Indeed, early experiments in microcomputer implementations confirm this prediction. From this initial observation, one is led inescapably to the hypothesis that a "distributed" data base

configuration may be desirable or appropriate in certain applications. Such a trend would undoubtedly lead to more careful consideration of communication between members of such a configuration, and it may become necessary to define the communication and global file sharing protocols engendered in such an environment.

Although the interpretive nature of MUMPS makes program development and modification easier than with many other languages, there is a cost in execution speed that may, in certain environments, be unacceptable. For this reason, there is a growing interest in "compiled" code generated by MUMPS systems. Such versions might require re-examination of features such as indirection as they currently exist in the language.

The MUMPS language does not lend itself to certain types of programming, such as scientific computation. An ideal solution to this weakness might be to implement MUMPS in a multilanguage environment, where its strengths could be complemented by those of other languages. Such environments have already been tested in limited installations. The experience this far gained is insufficient to predict trends in proposed changes to the language, but with additional experience some guidelines may emerge.

Summary

We have attempted to review briefly the manner in which MUMPS might change, as well as suggesting some possible directions these changes might take. It is important to stress, however, that any change to the language will be gradual, both as a result of the ANSI structure under which the MUMPS Development Committee operates, and as a consequence of the conservative attitude of the members of the Development Committee. Whatever changes may be proposed, they will be accepted only after careful evaluation, including actual testing in an operational environment, before they are adopted.

No computer language can remain stagnant in the face of man's growth in understanding of information management. What the designers of MUMPS hope, however, is that the evolution will be orderly and consistent with accepted change in computing as a whole.

Chapter 17

An Applications Example - Keyword-in-Context

Introduction

The previous chapters of the PRIMER have introduced the reader to the MUMPS language, its capabilities and limitations, and the features that distinguish it from other programming systems.

The purpose of this chapter is to present an applications example, complete with entry routines, calculations and outputs, together with a sample data set illustrating its use. The application that we have selected is one available (in expanded form) from the MUMPS Users' Group as an application package. The subset presented herein is the Keyword In Context (KWIC) approach to indexing references. KWIC index methods are useful in computer-generated bibliographies because they facilitate subject indexing without requiring subject classification by a content expert. The assumption is made that the titles are informative, and that indexing on the keywords within a title, especially if those keywords appear in the context of the original title, will be an acceptable alternative to detailed subject analysis.

The approach used in KWIC indexing is to take every word in the title of an article (excluding connective words such as and, the, of, from, etc.) and listing them in alphabetical order down the center of a page, giving the words that precede and follow each keyword in order, with the article's reference number at the end of the line. Such an index can be a helpful guide to searching for pertinent references within a bibliography. Compared with subject classification, which may require from ten to sixty minutes per article by a content expert, this approach is considerably less expensive at the cost of losing some subject precision.

MUMPS is ideally suited for such an application. The interpretive nature of the MUMPS language lends itself well to the development of such packages. Its interactive mode is ideal for the entry and update of bibliographic files, and its string manipulation features are well adapted to the problem of selecting and sorting keywords within bibliographic references. Comparable packages written in other languages would undoubtedly be more expensive to develop; one commercially available version written in COBOL was recently advertised as selling for over $600.00 (this chapter may therefore justfy the purchase price of the text by itself).

Design of KWIC Package

The functions required to create and maintain a KWIC index to a bibliographic file include the following:

1. Entry of new references
2. Editing and updating of existing references
3. Printing of the KWIC Index
4. Printing of the Full Bibliographic Citation

In order to concentrate on the more interesting aspects of the problem, we will limit our discussion to items (1) and (3) above. Editing the file and printing out the full bibliography, while necessary, may be accomplished by a number of techniques requiring varying degrees of sophistication. We will leave these elements to the reader to design.

System Driver Routine

Entry into a KWIC package of the type described above should be controlled by a single main entry routine that allows the user to select the activity he wishes to perform during that session. The modes included in this package at present are the following:

I = Input new data

L = List current file

K = Generate updated KWIC index

Q = Quit

(Note that no edit provision has been provided. We leave that task to the reader).

The following Routine should serve as an entry point driver for the KWIC package:

```
KWICMAIN; CREATED 10/78 BY RFW
     ;    USES GLOBAL FILE @REF (INDIRECTION TO ALLOW USER NAME)
     WRITE !,"THIS PACKAGE IS DESIGNED TO ACCEPT BIBLIOGRAPHIC"
     WRITE !,"REFERENCES, TO LIST THEM, AND TO CREATE A"
     WRITE !,"KEYWORD-IN-CONTEXT (KWIC) INDEX OF THEIR TITLES.",!
FILEQ READ !,"ENTER NAME OF FILE YOU WISH TO REFERENCE: ",REF
     SET REF="^"_REF IF $DATA(@REF) WRITE "FILE NOT FOUND" GOTO FILEQ
```

```
      WRITE !!,"YOU MAY SELECT ANY OF THE FOLLOWING OPTIONS:"
      WRITE !,?6,"I",?10,"= INPUT NEW DATA"
      WRITE !,?6,"L",?10,"= LIST CURRENT FILE"
      WRITE !,?6,"K",?10,"= GENERATE KWIC INDEX"
      WRITE !,?6,"Q",?10,"= QUIT"
START READ !,"PLEASE SELECT AN OPTION: ",CNTRL
      IF CNTRL?1"I".E DO ^BIBENTRY GOTO START
      IF CNTRL?1"L".E DO ^BIBLIST GOTO START
      IF CNTRL?1"K".E DO ^KWIC GOTO START
      IF CNTRL?1"Q".E QUIT
      IF $LENGTH(CNTRL)=1,("ILKQ"'[CNTRL) WRITE "SELECT I,L,K OR Q"
      GOTO START      ;END OF MAIN CONTROL ROUTINE
      ;END OF MAIN CONTROL ROUTINE
```

Data Entry

In order to permit updates to data entry, we must design a file which will be open-ended, accepting new entries and merging their information with the existing data base. We would also like to print out full references in alphabetical order by senior author. The file designed for this application will contain the following items:

1. The authors' names, beginning with the last name of the senior author (using the form last, initials, then initials and last names of subsequent authors: e.g. Carlson, L.D. and J.F. Green).

2. The year of publication (a four digit integer).

3. The full title of the reference.

4. The bibliographic citation (using an appropriate series of conventions not checked in this package).

5. A reference number (selected by the user) that will control sequence of output of the full references. This number should be unique within the file.

A Routine used to enter these data must go through the following steps:

1. Determine how many entries have been created in the file previous to this entry session.

2. Accept, in order, the five items of information identified above.

3. Create additional entries in the data file, calculating the output reference number as described above and saving the information for subsequent use.

The Entry Routine must also allow for user-controlled exit from the input session.

A two-level global file is used to hold the citations. It has the following characteristics:

Name of File: @REF (the use of indirection allows the user to specify his own file, so that the program may be used with more than one bibliography).

Elements:

1st level: an integer, beginning with 1, indicating the sequence of entry.

2nd level:

1. Author;
2. Title;
3. Year of Publication;
4. Citation;
5. Reference Number.

The following MUMPS Routine accomplishes the tasks described above.

```
BIBENTRY; CREATED 10/1978 RFW
     ;THIS ROUTINE ACCEPTS BIBIGRAPHIC CITATIONS (AUTHOR, YEAR,
     ;TITLE, CITATION, REFERENCE NO), AND PLACES THEM IN @REF(N,1-5)
     WRITE !!,"YOU ARE ADDING ENTRIES TO ",REF," LIBRARY.",!
     WRITE!,"PLEASE ENTER INFORMATION AS PROMPTED IN FOLLOWING
FORMAT"
     WRITE !,?5,"AUTH:  AUTHORS (LAST NAME, INITS, THEN INITS LAST)"
     WR"MULTIPLE LINES OK. NULL LINE ENDS AUTHORS."
     WR"NULL RESPONSE TO FIRST PROMPT ENDS INPUT MODE."
     WRITE !,?5,"TITLE: FULL TITLE (MULTIPLE LINES OK. NULL TO END)"
     WRITE !,?5,"YEAR:  YEAR OF PUBLICATION (4 DIGITS)"
     WRITE !,?5,"CIT:   BIB. CITATION (MULTIPLE LINES. NULL TO END)"
     WRITE !,?5,"REF: REFERENCE NUMBER USED TO SORT LISTING",!
     SET LIMIT=5000 ;MAXIMUM NUMBER OF REFERENCES (ARBITRARY)
     ;
     ;*** LOOP TO FIND END OF CURRENT GLOBAL FILE ***
     SET I=$NEXT(@(REF_"(0)"))-1 SET:I=-2 I=LIMIT GOTO:I>0 BIBENTER
     WRITE !,"FILE FULL. SEE PACKAGE MANUAL TO EXPAND LIMITS."
;
     ;*** LOOP TO OBTAIN NEW REFERENCES ***
BIBENTER READ !,"AUTH: ",X IF X="" QUIT ;EXIT POINT
     SET ^(I,1)=X
NXTTL READ !,"TITLE: ",TITLE
     SET ^(2)=^(2)_TITLE IF TITLE'="" GOTO NXTTL
RDYR READ !,"YEAR",Z IF Z'?4N WRITE " 4 DIGITS" GOTO RDYR
```

```
     SET ^(3)=Z
NXCIT READ !,"CIT: ",CIT
     SET ^(4)=^(4)_CIT IF 'CIT="" GOTO NXCIT
     READ !,"REF: ",Z SET ^(5)=Z,I=I+1 GOTO BIBENTR
     ;*** END OF ENTRY ROUTINE ***
```

Generation of KWIC Index

A Keyword-in-Context (KWIC) Index is created by examining each word in the title, eliminating those words that are insignificant, and then printing a line in which the keyword is near the center of that line, as shown in the output example at the end of this chapter. In order to create such a an index, we must first have a small "dictionary" of insignificant words, i.e., those that would not convey useful information in the sorted output. The following utility can be used to create a global file of "stopwords" (the technical term frequently used for these common terms).

```
STOPWORD ; CREATED 6/76 BY MJS
     ;CREATES OR ENLARGES A DICTIONARY OF STOPWORDS
     ;FILED ACCORDING TO WORD LENGTH
SWLOOP READ !,"ENTER NEW STOPWORD (NULL TO QUIT): ",WORD
     IF WORD="" QUIT
     SET L=$L(WORD) IF L>10 WRITE !,"MUST BE <=10 CHAR" GOTO SWLOOP
     SET ^SW(L)=^SW(L)_"/"_WORD GOTO SWLOOP
     ;END OF ROUTINE
```

When this routine is executed, each new word is entered and concatenated with other words the same length in the SW global file. The words will be separated by a slash (/) to avoid misidentification.

A second utility routine required for the KWIC indexing is a sort routine. This utility may be one already available as a component of the system; however, since we cannot assume the existence of such a general purpose routine, we reproduce below one that will be used in conjunction with this package.

```
HEAPSORT ;CREATED BY SLJ 9/78
     ;REQUIRES A LOCAL ARRAY A(I) OF KEYS, B(I) OF INDICES
     ;TO REFERENCED GLOBAL ARRAY. N ITEMS SORTED
HPSRT SET L=N\2,R=N
     IF L>1 SET L=L-1,X=A(L),Y=B(L) DO SIFT GOTO HPSRT+1
     IF R>1 SET X=A(R),A(R)=A(1),Y=B(R),B(R)=B(1),R=R-1
     IF  DO SIFT GOTO HPSRT+2
```

```
     QUIT ;END OF HEAPSORT ROUTINE
SIFT SET J=L
     SET I=J,J=2*I
     IF J>R GOTO ENDSIFT
     IF J<R IF A(J+1)]A(J) SET J=J+1
     IF X]A(J) GOTO ENDSIFT
     SET A(I)=A(J),B(I)=B(J) GOTO SIFT+1
ENDHEAP SET A(I)=X,B(I)=Y
     QUIT
     ;END OF HEAPSORT ROUTINE
```

Note that the technique used in this routine is the "heap sort" described in several computer texts such as Knuth's Art of Computer Programming vol 3, p. 145ff. This is a more efficient technique than the conventional "bubble" sort, and will therefore save time in a large sorting task.

Having provided the key utilities that are needed for the KWIC index, we can create the actual keyword-in-context using the following routine:

```
KWIC ; CREATED BY MJS AND RFW 10/76
     ;USES HEAPSORT AND ^SW GLOBAL FILE
     ; BEGIN BY CREATING LOCAL STOPWORD VARIABLE
     FOR I=1:1:10 SET:$DATA(SW(I)) SW(I)=^(I)
     SET BLKS=$JUSTIFY("",80) ;80 BLANKS
     ;
     ;GET NEXT TITLE, CREATE KWIC LINES
     SET I=0,J=0
KWTTL SET I=I+1,X=REF_"("_I_",2)",KWIN=@X,KWID=^(5) Q:$DATA(KWID)=0
     SET PCOUNT=1,KWLNG=$L(KWIN),NULLS=0,PTR=1,KO=0
     ;
     ;PROCESS EACH WORD IN TITLE
KWLOOP SET KWRD=$PIECE(KWIN," ",PCOUNT),KWL=$LENGTH(KWRD)
     SET PCOUNT=PCOUNT+1
     IF KWRD="" SET NULLS=NULLS+1 IF NULLS>2 GOTO KWTTL
     IF KWL<3 GOTO KWLOOP ;IGNORE ONE- AND TWO-LETTER WORDS
     IF KWRD'?1A.E GOTO KWLOOP ;WORD MUST START WITH ALPHA CHAR
     ; NOW SEE IF WORD MATCHES A STOPWORD
     IF KWL<10 IF $FIND(SW(KWL),KWRD)>0 SET KO=KO+1 GOTO KWLOOP
     ;CREATE KEYWORD LINE WITH KEYWORD IN CENTER
     ;L1-L2 IS LEFTHAND SEGMENT; R1-R2 IS RIGHTHAND SEGMENT
     SET PTR=$FIND(KWIN,KWRD,PTR),L2=PTR-KWL+1,R1=PTR,TB=0,LB=0
     IF L2<1 SET L1=0,L2=0 GOTO SETR2
     SET L1=L2-23 IF L1<1 SET L1=1
SETR2 SET R2=R1+(47-KWL) IF R2>KWLNG SET R2=KWLNG+1
     SET LB=24-(L2-L1),TB=(47-KWL)-(R2-R1) IF L1>0 SET LB=LB-1
     IF TB>0 SET TB=TB+1
     ;
     ;SAVE KEYWORD IN ^KW(J), KEYWORD LINE IN ^KWRD(J)
```

```
     SET J=J+1,^KW(J)=KWRD
     SET ^KWRD(J)=$EXTRACT(BLKS,1,LB)_$EXTRACT(KWIN,L1,L2)_KWRD
     SET ^(J)=^(J)_$EXTRACT(KWIN,R1,R2)_$EXTRACT(BLKS,1,TB)
     SET ^(J)=^(J)_"   "_KWID ;KEYWORD LINE NOW COMPLETE
     GOTO KWLOOP
     ;
     ;NOW SORT KWIC LINES
     FOR J=1:1 QUIT:$DATA(^KW(J)=0 SET A(J)=^KW(J),B(J)=J
     SET N=J-1
     DO HEAPSORT
     ;
     ;NOW LIST KWIC INDEX
     SET TITLE="KWIC INDEX OF "_REF_" FILE",PAGE=1,LINES=0,I=0
     DO HEADER
KWRITE SET I=I+1 IF I>N QUIT
     SET LINES=LINES+1 IF LINES>55 DO HEADER
     WRITE !,KWRD(B(I)) GOTO KWRITE
HEADER WRITE #,?30,TITLE,?67,PAGE,! SET PAGE=PAGE+1,LINES=0
```

Application Example

The Routines provided above are illustrated in the text that follows in all particulars except generation of the stopword list. The examples selected are four titles of references to MUMPS papers that might be of general interest to the reader of this PRIMER. The full interaction between user and the KWIC routines is provided. In order to make the output appear cosmetically more attractive, we will assume that lower case data may be entered at the terminal and printed in the final KWIC index.

```
>DO KWICMAIN
THIS PACKAGE IS DESIGNED TO ACCEPT BIBLIOGRAPHIC
REFERENCES, TO LIST THEM, AND TO CREATE A
KEYWORD-IN-CONTEXT (KWIC) INDES OF THEIR TITLES.

ENTER NAME OF FILE YOU WISH TO REFERENCE: MPSLIB

YOU MAY SELECT ANY OF THE FOLLOWING OPTIONS:
      I   = INPUT NEW DATA
      L   = LIST CURRENT FILE
      K   = GENERATE KWIC INDEX
      Q   = QUIT
PLEASE SELECT AN OPTION: I

YOU ARE ADDING ENTRIES TO MPSLIB

PLEASE ENTER INFORMATION AS PROMPTED IN FOLLOWING FORMAT
     AUTH:  AUTHORS (LAST NAME, INITS, THEN INITS LAST)
            MULTIPLE LINES OK. NULL LINE ENDS AUTHORS.
            NULL RESPONSE TO FIRST PROMPT PROMPT ENDS INPUT MODE.
```

```
      TITLE: FULL TITLE (MULTIPLE LINES OK. NULL TO END)
      YEAR:  YEAR OF PUBLICATION (4 DIGITS)
      CIT:   BIB. CITATION (MULTIPLE LINES. NULL TO END)
      REF:   REFERENCE NUMBER USED TO SORT LISTING
AUTH: Greenes, R.A., A.N. Pappalardo, C.W. Marble, G.O. Barnett
AUTH:
TITLE: Design and Implementation of a
TITLE: Clinical Data Management System
TITLE:
YEAR: 1969
CIT: COMPUTERS AND BIOMEDICAL RESEARCH, 2, pp. 469-485.
CIT:
REF: G001
AUTH: Munnecke, T., R.F. Walters, J. Bowie, C.B. Lazarus,
AUTH: D.A. Bridger
AUTH:
TITLE: MUMPS: Characteristics and Comparisons with other Languages
TITLE:
YEAR: 1976
CIT: MEDICAL INFORMATICS, 2:3 pp. 173-196.
CIT:
REF: M002
AUTH: Johnson, S.L., R.F. Walters
AUTH:
TITLE: MUMPS Implementations on Microprocessors
TITLE:
YEAR: 77
4 DIGITS
YEAR: 1977
CIT: MEDICAL INFORMATICS,2:4 pp. 257-267.
CIT:
REF: J003
AUTH: Brigham, C.R., J.D. Halvorsen, J. Zimmerman
AUTH:
TITLE: QUEST: A Teaching Program Driver
TITLE:
YEAR: 1976
CIT: JOUR. COMPUTER BASED INSTRUCTION, 3:2 pp. 42-50.
CIT:
REF: B004
AUTH: Bowie, J., G.O. Barnett
AUTH:
TITLE: MUMPS: An Economical and Efficient Time-sharing
TITLE: System for Information Management
TITLE:
YEAR: 1976
CIT: COMPUTER PROGRAMS IN BIOMEDICINE 6, 11-22.
CIT:
REF: B005
AUTH:
PLEASE SELECT AN OPTION: ?? SELECT I,L,K OR Q
PLEASE SELECT AN OPTION: K
```

```
                                KWIC INDEX OF MPSLIB FILE       1

                        MUMPS:  Characteristics and Comparisons with    M002
gn and Implementation of a      Clinical Data Management System         G001
MUMPS: Characteristics and      Comparisons with other Languages        M002
plementation of a Clinical      Data Management System                  G001
                                Design and Implementation of a Clini    G001
 QUEST: A Teaching Program      Driver                                  B004
                        MUMPS - An Economical and Efficient Time-sharin B005
 MUMPS - An Economical and      Efficient Time-sharing System for  I    B005
                Design and      Implementation of a Clinical Data Ma    G001
                     MUMPS      Implementations on Microprocessors      J003
nt Time-sharing System for      Information Management                  B005
 and Comparison with other      Languages                               M002
ntation of a Clinical Data      Management System                       G001
ing System for Information      Management                              B005
  MUMPS Implementations on      Microprocessors                         J003
                                MUMPS: Characteristics and Compariso    M002
                                MUMPS Implementations on Microproces    J003
                                MUMPS - An Economical and Efficient     B005
          QUEST: A Teaching     Program Driver                          B004
                                QUEST: A Teaching Program Driver        B004
                      QUEST: A  Teaching Program Driver                 B004
n Economical and Efficient      Time-sharing System for Information     B005
 and Efficient Time-sharing     System for Information Management       B005

PLEASE SELECT AN OPTION: Q
>
```

Summary

The example above illustrates the manner in which the programs listed in this chapter would process the user's input. Note that some refinements could be added. A "wraparound" keyword-in-context will treat each title as though it were on a cylinder, rotating it so that the beginning and end of the line are separated by an identifier, and reproducing as much of the title as will fit by adding text where space is available. In this approach, titles that appear on a KWIC index sorted by their first word have the end of the title wrapped around to the start of the line. Another cosmetic improvement might be to eliminate partial words from the KWIC listing. These programmatic changes can be accomplished with relative ease in the MUMPS language, and are left as challenges to the reader.

Chapter 18

Another Application Example - Mailing List Package

Introduction

In order to give the reader a different example of the way in which MUMPS can be used, we have selected an application package that is used to process mailing lists. The package provides for the creation and updating of such a list and for selective retrieval from the total list to generate mailing lists or labels for specific subgroups. An important feature of the package is the use of "flags" or identifiers, defined by the user, that can be used for selective printing of the entries. Flags may signify special conditions, e.g., membership on special committees, status of dues payments, or type of client. When the file is printed, the user may define boolean conditions based on these flags to select appropriate entries for output.

This type of application can be easily developed and updated with the MUMPS language. Its on-line updating capabilities are essential in many mailing lists requiring dynamic file maintenance, and its character handling and indirection capabilities offer important tools for implementing the flag functions. Since most enterprises need to process mailing lists, you may find that this package will provide you a practical way to learn more about the MUMPS labguage.

Design of the Package

The overall organization of the package is illustrated by Fig. 18.1. Each module was designed to be both easy to use and versatile in its choices. The OPTION DRIVER initializes the package and issues a prompt for one of the five top level functions. The ADD/EDIT and DELETE functions carry out routine maintenance of file entries. Both functions utilize the LOOKUP module to select particular entries. The FLAG EDIT Module maintains the directory of flag names. Output of the file is performed by the PRINT and MAILING LIST modules. PRINT reproduces all data in the file for user review. MAILING LIST produces a name and address list of selected entries for mailing labels or other applications.

A critical component of any package design is definition of the file structure and organization. For this package, we decided that, since output is usually done alphabetically, it would be best to

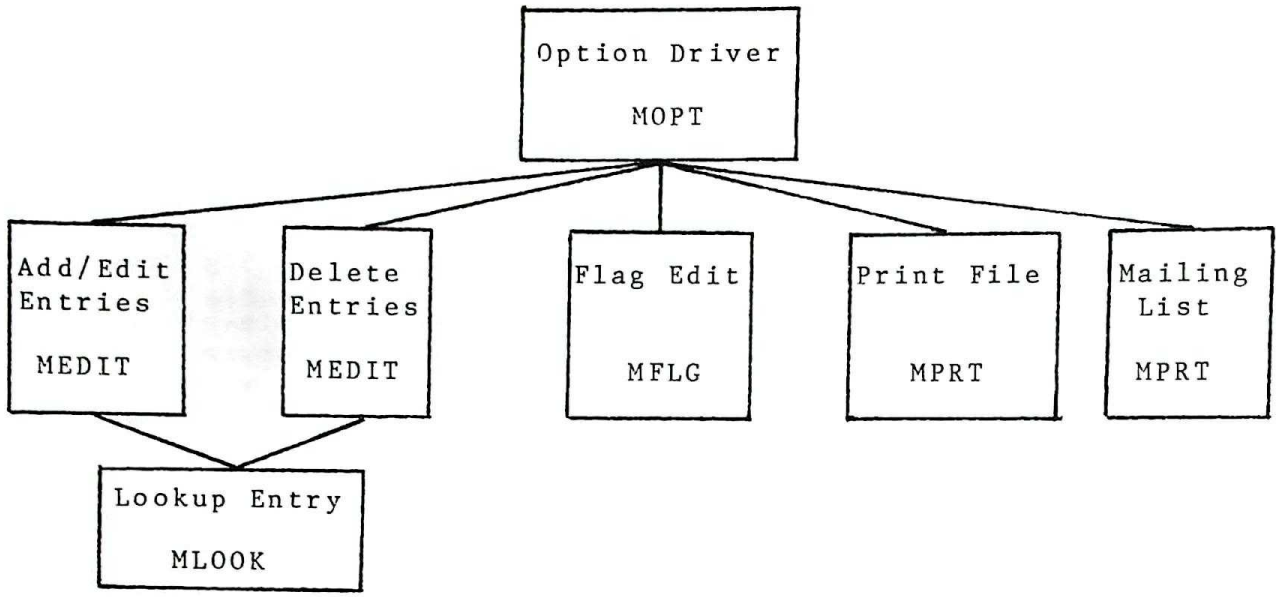

Fig. 18.1 Mailing List Package Organization Chart

Global File Organization

```
Main Entries File Name: ^MAIL

^MAIL(HASH,0) = Number of entries

    HASH =
      $ASCII(LASTNAME,1)*100+$ASCII(LASTNAME,2)*10+$ASCII(LASTNAME,3)

           ,1-9:      entry 1
           10-18:     entry 2
           ...

Line 0 =   entry name (last name first name [middle name][,titles]
Line 1 =   address (line 1)
Line 2 =   address (line 2)
Line 3 =   address (line 3)
Line 4 =   city
Line 5 =   state
Line 6 =   zipcode
Line 7 =   telephone number
Line 8 =   flag list


Flag Dictionary File

^FLAG($ASCII(FLAG)) = Description of Flag (arbitrary string)
```

Fig. 18.2 Global File Organization

maintain the file in alphabetical order. The name of each individual or firm in the file is therefore the primary key. Selection of individual entries is also by name. The lookup routine is included to avoid inadvertent duplicate entries.

Alphabetical ordering of the file is done by a bucket hashing technique. The first three characters of the last name are converted (hashed) to a number, the bucket index. All entries that fall in the same bucket will be kept in a single list in alphabetical order. This technique increases the efficiency of individual lookup and reduces the overhead (restructuring) necessary when new entries are added to the file.

Fig. 18.2 depicts the file organization. The number and meaning of each line in the entries can be adapted for individual needs. The flag list is a string of alphanumeric characters, each of which represents a certain condition or characteristic of the associated entry. The correspondence between a character and its meaning is stored in the flag dictionary. The FLAG EDIT module can be used to enter flag descriptions into the dictionary. Flags can be associated with entries in two ways, explicitly during entry through ADD/EDIT, or implicitly during a LIST function.

In the ADD/EDIT mode, multiple flags can be added to or deleted from specific entries. In the LIST operation, one or more flags may be added to all entries retrieved for that list. Using the LIST option for identifying certain entries, a special mailing list can be marked for one application. The remainder of the file could, at a later date, be sent the same mailing by using a not (') select condition on this flag, thereby avoiding duplicate mailings to the first group.

Description of Routines

The following sections describe the individual modules. They should be read in conjunction with the listings and sample interaction provided at the end of the chapter.

OPTION DRIVER

In many respects, the OPTION DRIVER is similar in function to KWICMAIN listed in the previous chapter. This module asks the user to select a choice, searches a routine table and performs the required option if a match is found. The user may input any number of characters to define an option. The driver used $FIND, $PIECE and $TEXT to determine if the input contains the initial characters of one of the options. If so, the remainder of the option name is

printed, and the second "piece" of the table entry is executed to perform the actual function. As with other queries, entry of a question mark lists the available options.

LOOKUP

LOOKUP accepts a name in standard format (first, middle, last, title), converts it into the internal format (last, first, middle, title) and then searches the main file for a match. If an exact match is not found, the program prints out a set of entries whose names are "close" to the one typed in. For the purposes of this package, "close" is defined as having the same last name (more sophisticated systems using SOUNDEX or other coding techniques could be substituted if a very large file were created). Pointers to the close entries are kept in the array IDX(I). If the user finds the name he is searching for among these entries, he may select it using both name and first address line. If none of the entries match the input, an ADD operation is assumed, and the routine searches the bucket for the appropriate insert location.

ADD/EDIT

The ADD/EDIT module is used both to enter a new name and also to edit the existing file entries. It is designed as a general query-response driver. The name of each entry line is typed first, followed by its current value in brackets, e.g.,

CITY [BOSTON]

The user may accept the current value by typing a carriage return, delete the value by typing a "-", or change the value by typing in a replacement. The responses are stored temporarily in the local array A for review prior to final acceptance of the modified fields for that record.

FLAG

Descriptions of the condition flags are stored in a global file ^FLAG (see fig. 18.2). The user may assign an arbitrary text string to any flag. The FLAG edit module accepts a flag name (which must be a single upper case alhabetic or a numeric character) and requests a descriptive string. If the flag already exists, its value may be edited. This module will also list the current flags and their descriptions by entering an asterisk (*) after the initial prompt.

PRINT and MAILING LIST

The PRINT and MAILING LIST functions are accommodated by the same module with two different entry points which perform the required initializations for each operation. Two parameters must be described for the MAILING LIST function, the "select condition" and the "update flags". The "select condition" is read in as a logical expression involving the flag names, then converted (label MAKB) to a valid MUMPS boolean expression. For example, if we wanted to get a list for all entries having flags A and B, or not having flag C, we would specify A&B!'C. This string would be converted to

```
BOOL=F["A"&(F["B")!(F'["C")
```

During processing, F will be set to the flag line of each entry. This condition is tested in the last line of the NEXT subroutine to select the candidate entries.

We noted earlier that entries can be assigned flags automatically during the MAILING LIST processing. These flags (UF) are read in at the line labelled LIST and added to the selected entries at the end of the BUFFER subroutine.

The PRINT function is very similar to MAILING LIST, but it does not include the select conditions or update flag option. PRINT initializes BOOL to 1 so that all entries will be printed. The differences in Print format and updating flags are processed at the end of BUFFER.

The output process itself is relatively straightforward, complicated only by a need to print multiple columns of data. The number of columns to be used is given in the aprameter COLS. This parameter can be changed to meet the needs of any specific label format. The main part of the routine is at LOOP. First, the entry buffer BUF is initialized. BUF will contain one entry for each column desired. The next alphabetic entry is selected by the subroutine NEXT. A subscript, S, is returned, pointing to the first line of the entry. For the LIST function, only entries passing the select conditions are returned. If no more entries exist, a negative value of S is returned. The subroutine BUFFER formats the entry into the BUF array. The telephone number and flag lines are output only by the PRINT function, while the update flags are only added for MAILING LISTs. Line LOOP+2 prints the output one line at a time with appropriate tabs to new label columns.

Listing of Routines

In this chapter, all the routines are listed together, so that they may be studied (or entered into your system) as a unit. The descriptive narrative for each appears in previous sections of this chapter. In this listing, individual command lines may extend beyond a single line of text. Remember, however, that a command line may be up to 255 characters long. When a comand line is too long for the output device, the system moves to the next line so as to continue printing; this carriage return is not in the command line itself and is not recognized as the end of the line when the code is executed.

MOPT Routine (Option Driver)

```
MOPT ;MAILING LIST OPTION DRIVER. CREATED BY JEB 1/79
     WRITE !,"MAILING LIST PACKAGE",!
     SET ESIZ=9,NL=7,LEN=35,COLS=2 ;ENTRY SIZE, NO. OF PRINT LINES, LI
NE LENGTH, COLUMNS
FN   READ !,"FUNCTION > ",FN QUIT:FN=""
     ;SEARCH FOR THE FUNCTION
     FOR I=0:1 QUIT:$TEXT(TABLE+I)=""  GOTO FOUND:$FIND($PIECE($TEXT(T
ABLE+I),";",2),FN)-1=$LENGTH(FN)
     WRITE " ?" GOTO FN ;DIDN'T FIND IT
     ;
FOUND SET X=$TEXT(TABLE+I) WRITE $EXTRACT($PIECE(X,";",2),$LENGTH(FN+1
,99),!
     SET FN=$EXTRACT(FN,1) ;FOR USE BY THE FUNCTIONS
     XECUTE $PIECE(X,";",3) GOTO FN
     ;
     ;HANDLE QUESTION MARK
QUES WRITE ! FOR I=1:1 QUIT:$TEXT(TABLE+I)="" WRITE ?10,$PIECE($TEXT(
TABLE+I),";",2),!
     QUIT
     ;
     ;OPTION DRIVER TABLE (MUST BE AT END OF PROGRAM)
TABLE ;?;DO QUES
     ;ADD/EDIT ENTRY;DO ^MEDIT
     ;DELETE ENTRY;DO DEL^MEDIT
     ;FLAG EDIT;DO ^MFLG
     ;PRINT FILE;DO PRINT^MPRT
     ;MAILING LIST;DO LIST^MPRT
```

MEDIT Routine (Add, Edit and Delete Functions)

```
MEDIT     ;ADD OR EDIT MAIN FILE. CREATED BY JEB 1/79
     WRITE ! DO ^MLOOK IF 'S KILL A,QRY,TEXT QUIT
     IF S>0 FOR I=0:1:ESIZ-1 SET A(I)=^(S+I) ;IF EDIT, GET ENTRY
     ELSE  SET A(0)=NAM FOR I=1:1:ESIZ-1 SET A(I)=""
     SET I=$PIECE(A(0),",",3),A(0)=$PIECE(A(0),",",2)_" "_$PIECE(A(0),
",",1)
     IF I'="" SET A(0)=A(0)_","_I ;ADD TITLE
     SET TEXT="NAME^ADDR1^ADDR2^ADDTS^CITY^STATE^ZIP^TELE^FLAGS" ;PROM
PT STRING
EDIT FOR I=0:1 SET QRY=$PIECE(TEXT,"^",I+1) QUIT:QRY=""  DO ELINE
REVIEW WRITE !!,A(0),! ;EXTERNAL NAME
     FOR I=1:1:3 WRITE:A(I)'="" A(I),! ;ADDRESS
     WRITE A(4),", ",A(5),"  ",A(6),! ;CITY, STATE, ZIP
     WRITE:A(7)'="" A(7),"  WRITE "[",A(8),"]",! ;TELE [FLAGS]
VERIFY READ !,"F(ILE), E(DIT), OR I(GNORE)? ",X
     GOTO FILE:X="F",EDIT:X="E",MEDIT:X="I" WRITE " ?" GOTO VERIFY
     ;
FILE IF S<0 SET S=-S,^(0)=^(0)+ESIZ FOR I=^(0):-1:S+ESIZ SET ^(I)=^(I-
ESIZ)
     ;
     ;NOW GET NAME IN INTERNAL FORMAT
     SET NAM=$PIECE(A(0),",",1) FOR I=1:1 QUIT:NAM'[" "  SET NAM=$PIEC
E(NAM," ",2,99)
     SET A(0)=NAM_","_$PIECE(A(0)," ",1,I-1)_","_$PIECE(A(0),",",2)
     FOR I=0:1:ESIZ-1 SET ^(S+I)=A(I) ;PUT DATA IN FILE
     GOTO MEDIT
     ;
     ;EDIT A PARTICULAR LINE
ELINE WRITE !,QRY," " WRITE:A(I)'="" "<",A(I)
     READ "> ",X QUIT:X=""
     IF $LENGTH(X)>LEN WRITE " [TRUNCATED] " SET X=$EXTRACT(X,1,LEN-1)
     GOTO UP:X="^",QUES:X="?" SET:X="-" X="" SET A(I)=X QUIT
     ;
UP   SET I=I-2 SET:I<0 I=99 QUIT
QUES FOR J=0:1:3 WRITE $PIECE($TEXT(QT+J),";",2),!
     QUIT
     ;
QT   ;ENTER A NEW VALUE OR
     ;     A NUL TO RETAIN OLD VALUE
     ;     A "-" TO DELETE CURRENT VALUE
     ;     AN "^" TO RETURN TO PREVIOUS QUERY
     ;
     ;ENTRY POINT FOR DELETE
DEL  DO ^MLOOK QUIT:'S
     GOTO DEL:S<0 ;NOTHING TO DO
     FOR I=S+ESIZ:1:^(0) SET ^(I-ESIZ)=^(I) ;DELETE THE ENTRY
     SET ^(0)=^(0)-ESIZ GOTO DEL
```

MLOOK Routine (Name Lookup)

```
MLOOK ;LOOKUP FILE ENTRY. CREATED BY JEB 1/79
      ;RETURNS S - POINTER TO ENTRY
      ;          <0 ENTRY NOT FOUND -S IS ENTRY LOCATION
      ;          =0 NULL ENTERED
      ;      NAM - ENTRY NAME (INTERNAL FORMAT)
      ;NAKED STATE SET
READ  SET S=0 READ !,"NAME > ",N QUIT:N=""  ;GET ENTRY NAME
      IF N="?" DO QUES GOTO READ
      ;FIND LAST NAME (LAST PIECE BY " ")
      SET NAM=$PIECE(N,",",1) FOR I=1:1 QUIT:NAM'[" "  SET NAM=$PIECE(N
AM," ",2,99)
      SET NAM=NAM_","_$PIECE(N," ",1,I-1)_","_$PIECE(N,",",2) ;INERNAL
FORMAT
      SET HASH=$ASCII(NAM)*100+$ASCII(NAM,2)*10+$ASCII(NAM3)
      IF '$DATA(^MAIL(HASH,0)) SET ^MAIL(HASH,0)=0,S=-1 QUIT
      SET N=^(0),LAST=$PIECE(NAM,",",1),I=0
      ;
      ;SEARCH FOR NAME, BUILD ARRAY OF CLOSE ENTRIES IN IDX(I)
      FOR S=1:ESIZ:N GOTO:^(I)=NAM FOUND IF $PIECE(^(S),",",1)=LAST SET
I=I+1, IDX(I)=S
      ;
      ;NOT FOUND, WRITE POSSIBLES
      GOTO NEW:'I ;NO CLOSE ENTRIES
      FOR J=1:1:I WRITE !,J,?5,^(IDX(J)),?45,^(IDX(J)+1)
WHICH READ !,"ENTER ENTRY NO. OR N(ONE) > ",X IF X="N" GOTO NEW
      IF X'?1N.N!(X>I) WRITE " ?" GOTO WHICH
      WRITE ! SET S=IDX(X) GOTO QUIT
      ;
FOUND WRITE "  ",^(S+1) ;WRITE ADDR1
      READ "   OK? ",X GOTO QUIT:X?1"Y".E,HERE ;DO ADD IF NOT OK
      ;
NEW   WRITE "  NOT FOUND",!
      IF N=0 SET S=1 GOTO HERE
      FOR S=1:ESIZ:N GOTO:^(S)]NAM HERE ;FIND CORRECT PLACE
      SET S=S+ESIZ ;PUT AT END
HERE  SET S=-S ;MARK AS NEW ENTRY
QUIT  KILL N,IDX,LAST QUIT
QUES  WRITE !,"ENTER NAME AS 'FIRST MIDDLE LAST,TITLE'",! QUIT
```

MPRT Routine (Print and List File)

```
MPRT ;OUTPUT ROUTINES. CREATED BY JEB 1/79
     ;REQUIRES COLS AND NL
     ;ENTER AT PRINT AND LIST
PRNT SET BOOL=1 GOTO OUT ;SELECT ALL AND DO OUTPUT
LIST SET BOOL=1 READ !,"SELECT CONDITION <1> ",X IF X'="" DO MAKB
UFLG SET UF="" READ !,"UPDATE FLAGS > ",UF
     IF UF'?.UN WRITE "UPPER CASE ALPHA OR NUMERIC ONLY" GOTO UFLG
OUT  R !,"OUTPUT DEVICE > ",IO IF IO="" SET IO=$I
     WRITE ! SET HASH=-1,(S,N)=0 USE IO WRITE # DO NEXT
LOOP IF S<0 WRITE !# USE 0 GOTO KILL
     FOR L=1:1:NL FOR C=1:1:COLS SET BUF(C,L)="" ;CLEAR BUFFER
     FOR C=1:1:COLS DO BUFFER,NEXT QUIT:S<0
     FOR L=1:1:NL WRITE ! FOR C=1:1:COLS WRITE ?C-1*40,BUF(C,L)
     WRITE !! GOTO LOOP
     ;
KILL KILL BOOL,UF,S,N,L,C,F,HASH,NAM,BUF,NOT,IO QUIT
     ;GET NEXT FILE ENTRY, S POINTS TO IT (<0 IF NONE)
NEXT SET S=S+9 GOTO NEW:S>N SET F=^(S+8) ;GET FLAG WORD
     IF @BOOL QUIT  ;NOTE MUST BE ARG. INDIRECTION
     GOTO NEXT ;TRY FOR ANOTHER
NEW  SET S=-8,HASH=$NEXT(^MAIL(HASH)) QUIT:HASH<0  ;NO MORE ENTRIES
     SET N=^(HASH,0) GOTO NEXT ;TRY THIS BUCKET
     ;
     ;FORMAT ONE ENTRY INTO BUF
     ;I IS OUTPUT LINE NUMBER
BUFFER SET I=1,NAM=^(S),BUF(C,I)=$PIECE(NAM,",",2)_" "_$PIECE(NAM,","
,1)
     IF $PIECE(NAM,",",3)'="" SET BUF(C,I)=BUF(C,I)_",",_$PIECE(NAM,"
",",3) ;HANDLE TITLES, IF ANY
     FOR J=1:1:3 IF ^(S,J)'="" SET I=I+1,BUF(C,I)=^(S+J) ;IGNORE NULL
LINES
     SET I=I+1,BUF(C,I)=^(S+4)_", "_^(S+5)_"  "_^(S+6) ;CITY,STATE, Z
IP
     IF FN="P" SET BUF(C,I+1)="" SET:^(S+7)'="" BUF(C,I,2)=^(S+7)_" "
     IF  SET BUF(C,I+2)=BUF(C,I+2)_"["_F_"]"
     ELSE  SET ^(S+8)=F_UF ;ADD UPDATE FLAGS
     QUIT
     ;
     ;MAKE A BOOLEAN CONDITION FROM 'X'
MAKB SET BOOL="",NOT="" FOR I=1:1:$LENGTH(X) SET C=$EXTRACT(X,I) DO C
HAR
     QUIT
     ;
CHAR IF C="'" SET NOT="" QUIT
     IF C?1UN SET BOOL=BOOL_"(F"_NOT_"["""_C_"""),NOT="" QUIT
     SET BOOL=BOOL_C ;MUST BE OPERATOR
     QUIT
```

MFLG Routine (Display and Edit Flags)

```
MFLG ;FLAG EDIT AND DISPLAY. CREATED BY JEB 1/79
READ READ !,"F;AG > ",FLG ;GET FLAG NAME
     GOTO LIST:FLG="*",QUIT:FLG=""
     IF FLG'?1UN WRITE " 1 ALPHANUMERIC CHAR ONLY" GOTO READ
     SET FA=$ASCII(FLG),X="" ;X IS CURRENT VALUE
     IF $DATA(^MFLG(FA)) SET X=^(FA) WRITE !,"<",X,">",!
READ !,">",Y,1 IF Y'="" SET:Y="-" Y="" SET ^MFLG(FA)=Y
     GOTO READ
     ;
     ;LIST THE FLAGS
LIST READ !,"OUTPUT DEVICE > ",IO IF IO="" SET IO=$I
     USE IO WRITE #!,?30"FLAG LISTING",!! ;72 COLUMN LINE ASSUMED
     SET FA=-1,X=$DATA(^MFLG(0)) ;SET NAKED
     FOR I=0:0 SET FA=$NEXT(^(FA)) QUIT:FA<0 WRITE $CHAR(FA),?5,^(FA
),!
     WRITE # USE 0 GOTO READ
QUIT WRITE ! KILL FLG,FA,X,Y,IO QUIT
```

The listings provided above contain a number of techniques used by experienced MUMPS programmers. You will probably find it useful to study them, then to enter the routines into your system, executing portions as you do so. Since MLOOK is a utility used in creating new entries to the file, you should begin by entering this routine first. Then, enter the routine MEDIT, which uses MLOOK to create to the file. With the experienced gained from these exercises, you will undoubtedly find occasion to refer back to chapters in the PRIMER to remind you of features that are used in these routines.

Sample of Output using Mail List Package

The pages that follow reproduce a sample dialog with MAILING LIST. When you have entered all the routines presented above correctly, you should be able to duplicate the dialog as shown.

(User entries underlined)

```
D ^MOPT

MAILING LIST PACKAGE

FUNCTION > ?

           ADD/EDIT ENTRY
           DELETE ENTRY
           FLAG EDIT
           PRINT FILE
           MAILING LIST

FUNCTION > ADD/EDIT ENTRY


NAME > ?
ENTER NAME AS 'FIRST MIDDLE LAST,TITLE'

NAME > JOHN SMITH
NAME <JOHN SMITH>
ADDR1 > HAL INC.
ADDR2 > BOSTON
ADDR3 >
CITY > MASS
STATE > MA
ZIP > 02217
TELE > 617-555-9876
FLAGS > 1


JOHN SMITH
HAL INC.
BOSTON
MASS,MA  02217 617-555-9876  [1]


F(ILE), E(DIT), OR I(GNORE)? E
NAME <JOHN SMITH>
ADDR1 <HAL INC.>
ADDR2 <BOSTON> MAIN STREET
ADDR3 >
CITY <MASS> BOSTON
STATE <MA>
ZIP <02217>
TELE <617-555-9876>
FLAGS <1>

JOHN SMITH
HAL INC.
MAIN STREET
BOSTON, MA  02217
617 555-9876
F(ILE),E(DIT),I(GNORE)? F
```

```
NAME > JOHN SMITH
1     SMITH,JOHN                              HAL INC.
ENTER ENTRY NO. OR N(ONE) > N  NOT FOUND

NAME <JOHN SMITH>
ADDR1 > SYSTEMS CONSULTING
ADDR2 > ROOM 142
ADDR3 > BROADWAY
CITY > NEW YORK
STATE > NY
ZIP >
TELE >
FLAGS >

JOHN SMITH
SYSTEMS CONSULTING
ROOM 142
BROADWAY
NEW YORK, NY
[]

F(ILE),E(DIT), OR I(GNORE)? F
NAME >
FUNCTION > PRINT FILE (Note: some other names were added previously)

OUTPUT DEVICE >

JACK BOWIE, SCD                         JOHN JONES, PHD
MAIL STOP #641                          1234 STARS END
MITRE CORP.                             NEW YORK, NEWYORK 12345
1820 DOLLEY MADISON BLVD.
MCLEAN, VA  22102                       212-555-1212  []

703-827-6667  [A1]


JOHN SMITH                              JOHN SMITH
HAL INC.                                SYSTEMS CONSULTING
MAIN STREET                             ROOM 142
BOSTON MA  02217                        BROADWAY
                                        NEW YORK, NY
617-555-9876  [1]
                                        []


RICHARD WALTERS, PHD
DEPT OF COMMUNITY HEALTH
SCHOOL OF MEDICINE
UNIVERSITY OF CALIFORNIA
DAVIS, CA  95616

916-752-2793  [A]
```

```
FUNCTION > FLAG EDIT

FLAG > 2
>RECEIVED SECOND MAILING

FLAG > *
OUTPUT DEVICE >
                          FLAG LISTING

1     RECEIVED FIRST MAILING
2     RECEIVED SECOND MAILING
A     AUTHOR

FLAG >

FUNCTION > MAILING LIST

SELECT CONDITION <1> A!'1
UPDATE FLAGS > 2
OUTPUT DEVICE >
```

JACK BOWIE, SCD
MAILSTOP #641
MITRE CORP.
1820 DOLLEY MADISON BLVD.
MCLEAN VA 22102

JOHN JONES, PHD
1234 STARS END
NEW YORK, NEWYORK 12345

JOHN SMITH
SYSTEMS CONSULTING
ROOM 142
BROADWAY
NEW YORK, NY

RICHARD WALTERS, PHD
DEPT OF COMMUNITY HEALTH
SCHOOL OF MEDICINE
UNIVERSITY OF CALIFORNIA
DAVIS, CA 95616

```
FUNCTION > ADD/EDIT ENTRY

NAME > SMITH,J
1     SMITH,JOHN,                                HAL INC.
2     SMITH,JOHN,                                SYSTEMS CONSULTING
ENTER ENTRY NO. OR N(ONE) > 2

NAME <JOHN SMITH>
ADDR1 <SYSTEMS CONSULTING>
ADDR2 <ROOM 142>
ADDR3 <BROADWAY>
CITY <NEW YORK>
STATE <NY>
ZIP >
TELE > 212-555-1212
FLAGS <2>

JOHN SMITH
SYSTEMS CONSULTING
ROOM 142
BROADWAY
NEW YORK, NY
212-555-1357  [2]

F(ILE), E(DIT), OR I(GNORE)? F
NAME >

FUNCTION > PRINT FILE

OUTPUT DEVICE >

JACK BOWIE, SCD                              JOHN JONES, PHD
MAIL STOP #641                               1234 STARS END
MITRE CORP                                   NEW YORK,NEWYORK  12345
1820 DOLLEY MADISON BLVD.
MCLEAN, VA  22102                            212-555-1212  [2]

703-827-6667  [A12]


JOHN SMITH                                   JOHN SMITH
HAL INC.                                     SYSTEMS CONSULTING
MAIN STREET                                  ROOM 142
BOSTON, MA  02217                            BROADWAY
                                             NEW YORK, NY
617-555-9876  [1]
                                             212-555-1357  [2]

RICHARD WALTERS, PHD
DEPT OF COMMUNITY HEALTH
UNIVERSITY OF CALIFORNIA
DAVIS, CA  95616

916-752-2793  [A2]
```

Summary

As you examine the output on the previous pages, note that the mailing list produced on the third page of output did not select the Boston Smith because his flags included [1], and the output specified that those people should be ignored in producing this list. Note, too, that the flags were updated so that all individuals except that individual had flag 2 added as a by-product of their appearing on that list.

The role of a PRIMER such as this one is to introduce you to the concepts of a language in such a way that you may familiarize yourself with its features in a logical, orderly fashion. There is, however, no substitute for practice when it comes to programming in any language. You will soon discover, therefore, that, even though you may have completed the PRIMER once, you will want to refer to it periodically as you confront new programming challenges. As you do so, refer to the Index of Language Elements just following the Table of Contents, so that you can quickly locate the desired feature. Periodically, review the last two chapters, and as you do so, try to understand a little more each time as to the techniques used. With a little experience, you should be able to improve these examples significantly (they were, after all, prepared primarily for the PRIMER and have not been tested in practice).

If your system has not prepared a supplement to this PRIMER to describe the language elements specific to that implementation, you may wish to consider preparing one yourself. There is no better way to learn about the language than to try to teach it to someone else. The authors of this text can attest to that from their own experience.

Acknowledgements

The application example presented in this chapter is based on a similar package developed by Robert Rees of the Laboratory of Computer Science, Massachusetts General Hospital. The authors are indebted to Bob for his helpful suggestions, especially in the area of flag operations, during the development of this chapter.

Chapter 19

Standard MUMPS Guide

INTRODUCTION

This section provides a concise summary of Standard MUMPS with explanatory text to provide a summary of MUMPS programming as a reference for users of the Primer. This section summarizes all the commands, functions, operators and other features of Standard MUMPS and is believed to conform to Standard MUMPS as defined by the MUMPS Development Committee and the National Bureau of Standards. However, the reader is strongly cautioned against using this Guide as a substitute for the official MDC/ANSI description of Standard MUMPS. For details of the Standard, consult: O'Neill, J. T., Editor, MUMPS Language Standard, ANSI X11.1-1977, published by the MUMPS Development Committee, 1977; or Conway, M. E., MUMPS Programmers' Reference Manual, published by the MUMPS Development Committee, 1976. For instruction in advanced programming techniques, the following is recommended: Beckley, R. E., and Bridger, D. A., Advanced Mumps Techniques, published by MUG, 1977.

To obtain copies or price lists of the MUG and MDC documents or to obtain copies of the Pocket Guide, contact Mr. Richard E. Zapolin, MUMPS User's Group, MITRE Corporation, P. O. Box 208, Bedford MA 01730.

Throughout this section, certain portions of the text are enclosed in brackets. This convention denotes certain limitations imposed not by the definition of Standard MUMPS, but by concern for the portability of MUMPS programs from one implementation to another. These Portability Requirements should be met in order to facilitate such interchange.

ACKNOWLEDGMENTS

This section is taken from the Pocket Guide second edition, compiled by Joel Achtenberg as an expansion and revision of the original Pocket Guide written by Joan Zimmerman under grant number HS-01540 from the National Center for Health Services Research, Department of Health, Education and Welfare. Special thanks in the revision were due to Dan Schullman for his extensive commentary and to Joan Zimmerman for her continued assistance. Substantial contributions to both editions were received from many members of the MUMPS community, particularly Jack Bowie, Robert Greenfield, Dan Schullman, David Sherertz, Robert Stimac, Tony Wasserman, and Jerry Wilcox.

This section is copyright by the MUMPS User's Group, September, 1978

DATA TYPES AND VALUES

The first thing to consider is the types of data which may be manipulated in Standard MUMPS. The situation is quite simple because there is only one data type: the variable-length character string, which may consist of any of the 128 ASCII codes. [To insure portability, the current limit on MUMPS string length is 255 characters.]

A number is regarded merely as a special case of a string, and MUMPS contains a well defined rule for interpreting any string as a number. A numeric value (that is, the value of a number) may contain a decimal fraction. However, some arithmetic operations produce an integer value, which is a special case of a numeric value. Accordingly, MUMPS also contains a rule for interpreting any number (and, by inference, any string) as an integer. Relational and logical operations (see below) produce a special numeric value called a truth value. There are two truth values: the integer 0, which denotes "false"; and the integer 1, which denotes "true". The interpretation rules are discussed under OPERATORS, below.

[To insure portability the value of a number should have no more than nine significant digits. The absolute value should lie between 10^{25} and 10^{-25}, or be zero.]

VARIABLES

A variable is an entity whose value may be changed. There are three variable types: local variables, each of which is unique to a user and whose value may be inspected and/or changed only by that user; global variables, each of which may have its value inspected and/or changed by any authorized user; and special variables, whose values are changed by the MUMPS system and cannot be changed directly by a user.

Each variable is referenced by a variable name. Local and global variable names begin with either "%" or an alphabetic. Subsequent characters may be any of the alphabetics or any of the ten digits. Names may be any length. [To insure portability, however, names should be distinguishable by their first eight characters and should not contain lower-case alphabetics.] A global variable name is designated by a leading caret (^) symbol, as in^ MUG. A special variable name is denoted by a leading dollar ($) symbol, as in $TEST. Special variables are discussed in a section below.

Local and global variables may be subscripted in order to facilitate the grouping of values into sets (called arrays). For example, it might be more convenient to replace the variable names GAME, SET, and MATCH by A(1), and A(2), and A(3) respectively. In this case, only one level of subscripting is used, but more can be used if desired. For example, A(1,4,13) has three levels of subscripting. Both A(1) and A(1,4,13) are said to be descendants of A, and A(1,4,13) is also a descendant of A(1).

Variables can have any number of subscripts. Subscripts may be any of the allowed non-negative integers (that is, from 0 to 999999999).

A naked global reference is a shorthand syntax for specifying a global variable by omitting the variable name and possibly some of the subscripts. The first subscript in the subscript list of a naked global reference implicitly refers to the last subscript level of the most recent global reference. Thus, if a reference has been made to ^X(1), a subsequent naked reference to ^(2,3) would access the value of ^X(2,3). Note that "last global reference" includes any reference to any global. Use of the $NEXT function is particularly troublesome as it may leave the naked pointer with an unexpected value. (See $NEXT, below, for additional discussion).

LITERALS

A literal has a constant value and is interpreted directly. There are both numeric and string literals. A numeric literal has a mantissa optionally followed by the letter "E" and an exponent (e.g. 0.34, 10E5, 1.2E-7, -1456). A string literal consists of a set of characters enclosed in quotation marks. The empty string is represented by two adjacent quotes (""). If quotes are to be embedded within a string, each must be represented by two adjacent quotes ("THE ""KNOWN"" VALUE"). Such embedded quotes are counted as one character, not two, when determining string length. A string literal consists of zero or more of the 95 ASCII graphics enclosed in quotes, whereas a string value may contain any of the 128 ASCII characters.

OPERATORS

The operators in Standard MUMPS are grouped into seven types, as follows:

Arithmetic Unary Operators
Arithmetic Binary Operators
Arithmetic Relational Operators
String Binary Operators
String Relational Operators
Logical Operators
Indirection

These groups differ in the interpretations made of their arguments. Each of the arithmetic operators takes the numeric interpretation of its arguments before performing the indicated operation. This involves taking the leftmost portion of the string which is either exponential (e.g. 86E5), decimal (-182.45) or integral (1964) in form. If no such form begins the string, the numeric interpretation is zero. The integer interpretation of any value is formed from the numeric interpretation by dropping any fraction. The logical operators interpret their arguments by first forming the numeric interpretation, if the result is 0 the interpretation is "false", otherwise, it is "true". The string operators require no special interpretation of their arguments since all data in Standard MUMPS are strings.

STRING	INTERPRETATION		
	NUMERIC	INTEGER	TRUTH VALUE
"810"	810	810	1
"98 POUNDS"	98	98	1
"" (the empty string)	0	0	0
" 35"	0	0	0
"86+9"	86	86	1
"PAGE 10"	0	0	0
"-8.4"	-8.4	-8	1
"86E-1"	8.6	8	1
"---9"	-9	-9	1
"-0"	0	0	0

Each of the Arithmetic, String and Logical operators is described, along with examples, in the table on the following pages. Indirection, although formally an operator, is somewhat unique in its use, and is therefore discussed below in a separate section.

INDIRECTION

Indirection allows data values to be interpreted as MUMPS code. Indirection is denoted by the character @ followed by an expression. The value of the expression is substituted for the occurrence of indirection before the rest of the line is interpreted. The substitution is temporary, taking place each time the instance of indirection is encountered. This allows the same segment of MUMPS code to be executed repeatedly with differing values of the expression yielding different results. Furthermore, indirection can be nested, giving even greater flexibility. There are three basic types of indirection:

In Argument Indirection the indirection occurs in place of a command argument, and the value must be one or more complete command arguments.

```
ex.  S X(1)="STATE(I)",X(2)="CAPITAL(I)",X(3)="POPUL(I)"
     F I=1:1:50 F J=1:1:3 W $P(X(J),"(",1),": " R @X(J),!
```

reads the name, capital and population of each of the fifty states and stores them in arrays STATE, CAPITAL and POPUL.

```
ex.  S PRINTER=3,TERMINAL=0
     R !,"DEVICE TYPE? ",DEV OPEN @DEV
```

In this example, the user $enters "PRINTER" or "TERMINAL" and the proper device will be opened.

```
ex.  S USERCODE="B"
     S @("AA"_USERCODE_"=99")
```

In Name Indirection the indirection occurs in any context where a named variable can occur and the value of the indirection must be a complete variable name, possibly including subscripts.

Table of Operators

Operator Symbol	Meaning	Examples	Results & Comment
ARITHMETIC UNARY OPERATORS			
+	Takes the numeric interpretation	+2 +"34A"	2 34
-	Takes the numeric interpretation and negates it	-A -"34A"	-7 if the value of A is 7 -34
ARITHMETIC BINARY OPERATORS			
+	Produces the sum	2+7 A+3	9 The value of A plus 3
-	Produces the difference	2-7 A-B	-5 The difference between A and B
*	Produces the product	2*7	14
/	Produces the full division	7/4	1.75
\	Division with the result truncated to an integer	7\4 A\5	1 0 where A is greater than -5 but less than 5
#	Produces the value of the left argument modulo the right argument (i.e. it gives the remainder)	11#3 11#-3 -11#3 -11#-3	2 -1 1 -2
ARITHMETIC RELATIONAL OPERATORS			
<	Less than	MINIMUM < 8	1 if the numeric value of MINIMUM is less than 8; 0 otherwise
>	Greater than	MAXIMUM > LIMIT	1 if the numeric value of MAXIMUM is greater than the numeric value of LIMIT; 0 otherwise

Table of Operators (continued)

Operator Symbol	Meaning	Examples	Results & Comment
STRING BINARY OPERATOR			
_	Concatenates	"GO"_END	"GOING" if END="ING" "GOES" if END="ES" "GO" if END=""
STRING RELATIONAL OPERATORS			
=	Equals	NAME="JONES"	1 if the value of NAME is "JONES"; 0 otherwise
[	Contains	NAME["SON"	1 if the value of NAME contains "SON"; 0 otherwise
]	Follows	NAME]"M"	1 if the value of NAME follows "M" in the ASCII collating scheme; 0 otherwise
?	Pattern matches	NAME?2A	1 if the value of NAME contains exactly two alphabetics; 0 otherwise
	For the pattern-match operator the allowed pattern codes are: A (for the 26 upper-case and 26 lower-case alphabetics); C (for the 33 control characters); E (for the entire set of 128 characters); L (for the 26 lower-case alphabetic characters); N (for the 10 numeric characters); P (for the 33 punctuation characters); U (for the 26 upper-case characters); or any string literal.	NAME?1A.A	1 if the value of NAME has one or more alphabetics; 0 otherwise

Table of Operators (concluded)

Operator Symbol	Meaning	Examples	Results & Comment
	The number of occurrences of each pattern type may be specified exactly by a preceding integer, or may be left inexact by a preceding ".". Thus, "1N" checks for exactly one numeric character and ".P" checks for any number of punctuation marks, including none. Allowed patterns may be combined into groups. For example, while ".A" checks for alphabetics and ".P" checks for punctuation, ".AP" checks for both alphabetics and punctuation; note that ".PA" has the same effect. However, literals may not be combined with pattern codes (i.e. .A"7" is not allowed).		
LOGICAL OPERATORS			
&	And	A&B	1 if A and B are true 0 if A and B are not both true
!	Or	C!D	1 if C or D is true 0 if neither C nor D is true
'	Not	'E	1 if E is false 0 if E is true
	Note that not (') may be used in conjunction with any arithmetic relational or string relational operator (e.g. A'=3, which is the same as '(A=3)).		

```
ex.  S AGE=40,RACE="WHITE",SEX="FEMALE"
     R !,"WHAT VARIABLE WOULD YOU LIKE TO SEE? ",X
     W !,X," HAS THE VALUE ",@X
```

yields RACE HAS THE VALUE WHITE if the user answers the question with "RACE".

In Pattern Indirection the indirection occurs in place of a pattern, and the value must be a pattern.

```
ex.  S X(1)="1""$""1N.N",Y(1)="MONEY"
     S X(2)="5U.U",Y(2)="WORD"
     FOR I=1,2 IF STRING?@X(I) WRITE !,Y(I) Q
```

yields "DOLLARS" if STRING is one or more digits preceded by a dollar sign, or "WORD" if STRING is five or more upper-case letters.

Note that the XECUTE command (see below) also provides a means of performing indirection.

EXPRESSIONS

The simplest expression in MUMPS is a variable, a string literal, a numeric constant, or a function (functions are discussed in a section below). Examples of each of these four types of expression are respectively:

```
VARIABLE
"LITERAL"
45.73
$LENGTH(XYZ)
```

Such simple expressions are called atomic expressions. More complicated expressions can be built up by linking a number of atomic expressions by means of the arithmetic and other types of operator. For example:

```
SUM/TOT
SEX="MALE"
"BOY"_"HOOD"
```

All MUMPS expressions are evaluated from left to right. There is a hierarchy of operators: unary operators are executed before indirection, which is executed before binary and relational operators. There is, however, no hierarchy among binary and relational operators. Parentheses can be used to modify the order of evaluation.

COMMANDS

A command defines an action to be taken. A command is usually (but not always) followed by an argument (or a series of arguments separated by commas) upon which the command acts. Most command words may be abbreviated to their initial letter, or may be fully spelled out. Note that partial abbreviations are not allowed. Thus, "B" is a legal abbreviation for "BREAK" but "BR" is not. All unspecified initial letters are reserved for future use.

Command	Examples	Explanation
BREAK	B	BREAK provides an access point within MUMPS for direct mode (nonstandard) programming and debugging aids. It suspends execution until receipt of a signal (not specified) from a device. BREAK does not normally take an argument, and any arguments used are implementation specific.
CLOSE	C 4 C MT1:(devparms)	CLOSE releases the listed devices from ownership; for example, it releases device 4 or MT1. A list of implementation-specific device parameters may be placed after a device name if desired. If the current device is closed, the special variable $IO will be empty or reset to a default value, depending upon the implementation.
DO	D NEW,OLD D ^A D LINE^ROUTINE D @VARIABLE	Each of the line labels (for example, NEW and OLD) listed in the argument are found in turn and the code following each label is executed until the end (specified below) is reached. If "^" precedes a label (as in the case of ^A), the routine whose name (A) follows the "^" character is executed. Execution starts at the first line of the named routine unless a line label (for example, LINE) is specified in front of the "^". In the latter case, execution starts in the routine named ROUTINE at the line labeled LINE. Execution returns immediately to the next argument or the next command after executing a QUIT command or reaching the end of a routine (provided this was not within the scope of a subsequent DO command or a FOR command). Indirection (e.g. "@VARIABLE") can be used to begin execution of the line, or routine whose label is contained in a variable.
ELSE	E	ELSE permits conditional execution dependent upon the truth value of the special variable $TEST (discussed below). Execution of the remainder of the line to the right of the ELSE command continues only if the value of $TEST is 0 (false). If $TEST is 1 (true) any commands to the right of ELSE are not executed. ELSE does not take an argument.
FOR	F I=7,"BD" F J=2:4:20 F K=1:1 F L=10:-1:0,13,15:1	FOR specifies the repeated execution of all the commands following it on the line. There are three formats. First is the expression format, where a local variable is assigned a value such as the value 7, or "BD". Second is the range format, where a local variable is set to an initial value and then incremented in specified steps until it reaches or exceeds a limiting value; in the second example, the initial value is 2, the incremental step is 4, and the final value is 20. Note that in this case 18 is the last value of J for which the associated commands will be executed. Third is the "endless" loop format wherein only an initial value and an incremental value are specified, as shown in the third example; the "endless" loop is broken only by execution of a QUIT or a GOTO command within the instructions following the FOR command. Any of the above formats may be mixed; in the final example, we show the range format (using a negative increment), then the expression format, and finally the "endless" loop. Note that a QUIT or GOTO command terminates all forms of the FOR loop, not just the "endless" loop, and that a QUIT will terminate only the most recent in a series of nested FOR loops, whereas GOTO will terminate all active FOR loops.

Table of Commands (continued)

Command	Examples	Explanation
GOTO	G NEXT G ^ROUTINE G LINE^ROUTINE G @VARIABLE	GOTO transfers execution of code (without return) either to a line of code in the same routine (for example, G NEXT), to the start of a specified routine (for example, G ^ROUTINE), or to a particular line of code in a specified routine (for example, G LINE^ROUTINE causes execution to be transferred to the routine named ROUTINE at the line named LINE). (Note that if return of control is required, the DO command should be used.)
HALT	H	First LOCK (see below) with no arguments and CLOSE of all devices opened by this job are executed (although these are not stated explicitly). Then execution of the current process is terminated. HALT does not take an argument.
HANG	H 3 H TIME	HANG suspends execution for the number of seconds specified by each argument (such as 3 or TIME). If an argument is less than zero, execution is suspended for zero seconds. Note that HANG without an argument becomes HALT.
IF	I I A=3 I B="B",A>4 I C=4! (C=5) I $D(X),X=4	IF permits conditional execution. In its argumentless form, execution is dependent upon the value of $TEST. If the value of $TEST is 0 (false), the remainder of the line to the right of the IF is not executed; if the value of $TEST is 1 (true), execution continues normally at the next command on the line. If one argument is present (such as A=3), the argument is evaluated to be true or false and $TEST is set to 1 or 0 respectively; then the IF command is executed as in the argumentless form. A series of arguments following an IF command is treated like a series of IF commands with single arguments, the ultimate result being the same as if the arguments were "and"-ed together, except that as soon as one false argument is found, interpretation and execution of the line ceases.
KILL	K K A,NEW,^P(3) K (SAVE,A)	In its argumentless form, KILL permanently removes all existing local variables, including subscripted local variables. When an argument is specified, the local and global variables named in the argument are deleted together with all descendant variables. In the second example, local variables A and NEW are removed, as are all nodes in global ^P which have a first-level subscript of 3. The "exclusive kill" is a special form of the KILL command where the argument of the KILL command is a series of unsubscripted local variables (such as SAVE and A) contained in parentheses; all local variables are killed except those named and their descendants.

Command	Examples	Explanation
LOCK	L L ^MUG L (A,B,^G(4)) L X123:0	LOCK is used primarily as a software convention to avoid conflicting updates of named resources (primarily global variables). In executing each argument, LOCK releases all previously specified exclusive claims to resources. If any arguments are specified, new temporary exclusive claims are established to the named resources. If another user has exclusive claims on a resource named, the current user (that is, the one executing the LOCK command) remains suspended awaiting the availability of that resource. A "timeout" can be affixed to any argument of LOCK to abort an unsuccessful wait, as mentioned above. (Note that a series of named resources must be placed in parentheses, as for L(A,B,^G(4)) but that parentheses are not required around the name for a single resource.) LOCK has no effect on the value or definition of existing variables or on the "naked indicator."
OPEN	O 4 O 7:(devparms)	The OPEN command is used to obtain exclusive ownership of a device. It does not affect the "current device" with which the routine is interacting. Implementation-specific device parameters may be placed after a device name. Ownership is relinquished upon execution of the CLOSE command. A "timeout" may be affixed to any argument to abort an unsuccessful attempt to OPEN that device.
QUIT	Q	QUIT defines an exit point following a FOR, a DO, or an XECUTE command. It does not take an argument.
READ	R VALUE,X R *A R "NAME: ",N R "WAIT? ",X:30	READ calls for data input from the current device, and the assignment of the response to a named local variable. When the argument contains an asterisk preceding a variable name, a code representing a single character is obtained. Any text and format control characters (see below) in the argument of the READ command are output on the current device. The amount of time for completion of input may be specified by affixing a timeout to an argument of READ.
SET	S NEW="BIG" S X="B",^GO(1)=9 S (I,J,K)=1	Set is the general means for explicitly assigning values to variables. When a list of variable names in parentheses is placed between the SET command and the assignment symbol ("="), each named variable is given the value following the assignment symbol.
USE	U 4 U MT+2:(devparms)	USE designates a specific device (such as device number 4 or device MT+2) as the current device for input and output, or device status. Device designators are implementation specific. Before a device can be named in the argument of a USE command, its ownership must have been established through execution of an OPEN command. As for CLOSE and OPEN, implementation-specific device parameters may be placed after a device name.

Table of Commands (concluded)

Command	Examples	Explanation
VIEW	Implementation specific	VIEW provides an access point within Standard MUMPS for the examination or modification of implementation-dependent information.
WRITE	W "HELLO" W A,B,Z W *13 W !?5,NAME W "AVG= ",SUM/TOT	WRITE specifies the output of data and format control to the current device. When an argument includes an asterisk followed by an integer value, one character whose code is the number represented by the integer is sent to the current device. The effect of this character at the device is device-dependent and implementation specific.
XECUTE	X "S A=3"	XECUTE provides a means of interpreting MUMPS code which arises during program execution. Each argument of the XECUTE command is interpreted as if it were a line of MUMPS code (without label or line start indicator).
Z	Implementation specific	Names of commands beginning with the letter Z are reserved for implementation specific extensions to Standard MUMPS.

A command and its arguments in a line of MUMPS code are separated from the next command (if any) on that line by a single space. Also, a command word is separated from its arguments by a single space. Note that this means an argumentless command is separated from the next command by exactly two spaces. In general, for any command Z with arguments A and B, Z A,B is equivalent to Z A Z B. Comments can be appended to a line by placing a semicolon in any command position (i.e., wherever a command could go). The remainder of the line is treated as a comment and is not executed.

TIMEOUTS ON COMMANDS

A timeout may be used with arguments of the LOCK, OPEN, and READ commands to specify the maximum time during which MUMPS will await the completion of the associated operation. If a timeout is to be used, it is specified by following an argument of any of these commands with a colon and then an expression whose value is an integer. For example:

```
READ ANSWER:TIME
```

will wait for up to TIME seconds to obtain ANSWER. If the numeric value following the colon is positive, that value specifies the maximum number of seconds waited for the completion of the operation. If the numeric value following the colon is zero or negative, execution continues without delay. Note that if a LOCK, OPEN, or READ command with a timeout specification is satisfied before the time runs out, the special variable $TEST (discussed below) is set to 1; otherwise $TEST is set to 0. If no timeout is specified, $T is not affected and execution of the command proceeds after the condition associated with the command (for example, termination of the input message for the READ command) is satisfied. Therefore, execution can be suspended indefinitely.

POST-CONDITIONALS ON COMMANDS AND ARGUMENTS

The IF command may be used to place a condition upon whether or not the remainder of the line following the IF command is executed. Alternatively, an individual condition may be placed upon most commands by appending a colon and then a truth-valued expression to a command word. This may be done for all commands except ELSE, FOR, and IF. For example,

```
SET:A=3 NEXT=47 GOTO LINE
```

means that NEXT has the value of 47 assigned only if A has the value 3, but the GOTO command is always executed. The first command is said to be post-conditionalized. It is executed only if the expression immediately following the command and colon is true. Note that unlike the IF command, only the post-conditionalized command is affected; all commands to its right are still executed as they would be normally. Also, the value of $TEST is not affected.

The commands DO, GOTO, and XECUTE permit post-conditionalization of arguments and/or post-conditionalization of the command itself. This means that an argument of any of these commands may be followed by a colon and then a truth-valued expresssion. If the expression is true, that argument is used. Otherwise, it is not. For example:

```
DO FIND:A=100,NEXT
```

will execute the code beginning at the line labeled FIND only if A equals 100, but will always execute the code beginning at the line labeled NEXT. Thus, only an argument which has been post-conditionalized is affected. All arguments and further command-argument pairs to the right of a post-conditionalized argument are executed as they would be normally.

The following example illustrates post-conditionalization of both command and argument:

```
DO:B="OK" FIND:A=3,NEXT
```

FUNCTIONS

Each function is designated by an initial character of "$", and has a unique name which may be abbreviated to its initial letter. This specification of the function is followed by one or more expressions in parentheses. The first expression generally signifies the string or number which is to be examined or manipulated. Any subsequent expressions qualify the function's effect. All unspecified initial letters are reserved for future use.

Some functions restrict the type of expressions that can be used. The following codes will be used to indicate such restrictions. In each case "n" represented the position in which the expression is used.

En	general (unrestricted) expressions
In	Integer-valued expressions
Tn	Truth-valued expressions
Nn	Numeric valued expressions
L	Labels
VN	Variable names

SPECIAL VARIABLES

Each special variable is denoted by the initial character of "$" and has a unique name which may be abbreviated to its initial letter. The value of a special variable may be used as part of any general expression. Note, however, that a user is not permitted to assign a value to any special variable. All unspecified initial letters are reserved for future use.

Special Variable	Explanation
$HOROLOG	$HOROLOG provides the date and time in a single two-part string. The two parts are separated by a comma. The first part is the number of days since 31 December 1840, and the second part is the number of seconds since midnight. "0,0" is the first second of 31 December 1840.
$IO	$IO provides the unique identification of the current input/output device.

Table of Functions

Function Name and Syntax	Explanation	Examples	Results
$ASCII(E1) $ASCII(E1,I2)	$ASCII selects a character of a string and returns its ASCII code. E1 specifies the string of interest. If I2 is not specified explicitly, it is assumed to have a value of 1. The character in position I2 of string E1 is extracted and translated into the integer which represents it in the ASCII sequence.	$A("ABC") $A("ABC",1) $A("ABC",2) $A(X,2) $A(" ")	65 65 66 66 where X="ABC" -1
$CHAR(I1) $CHAR(I1,I2) $CHAR(I1,I2,...)	$CHAR translates a set of integers into a string of characters whose ASCII codes are those integers. I1,I2 and so on are the expressions whose values are interpreted as the integers (which must be in the range 0-127, inclusive). A negative integer yields an empty string; values greater than 127 yield an error.	$C(65) $C(66) $C(65,66)	"A" "B" "AB"
$DATA(VN)	$DATA returns an integer specifying whether a variable is defined or undefined. VN is the name of the local or global variable of interest. The values returned are 0 if VN is not defined, 1 if VN has a value but has no associated descendant array members, 10 if VN does not have a value but has at least one descendant and 11 if VN has both a value and at least one descendant. Note: VN might not have a descendant even when $D(VN) > 9 under certain circumstances; e.g. all descendants have been killed.	$D(Y) $D(Y) $D(^A(1,5))	0 if Y is undefined 1 if Y has a value but has no descendant 0, 1, 10 or 11
$EXTRACT(E1,I2) $EXTRACT(E1,I2,I3)	$EXTRACT returns a character or substring of string E1 beginning at character position I2 and ending at character position I3 (at I2 if I3 is not specified) or at the end of the string, whichever occurs first.	$E("ABC",2) $E(VAR,3,5) $E(VAR,3,999) $E(S,90) $E("XYZ",-6,2)	"B" "CDE" if VAR="XXCDE". "10A" if VAR="B910A". " " if S is less than 90 characters long. "XY"

Table of Functions (continued)

Function Name and Syntax	Explanation	Examples	Results
$FIND(E1,E2) $FIND(E1,E2,I3)	$FIND returns an integer which specifies the end position plus one of a substring (E2) within a string (E1). If a third expression, I3, is specified, the search in E1 for E2 begins at the character in position I3. Otherwise the search begins at position 1. If the portion of E1 beginning at I3 (or 1) is the empty string or does not contain E2 the result is 0 (zero). If E2 is null (but not the portion of E1 beginning at I3 or 1) the result is I3.	$F("ABC","B") $F("ABCABC","A",3)	3 5
$JUSTIFY(E1,I2) $JUSTIFY(N1,I2,I3)	$JUSTIFY with 2 arguments returns the value of expression E1, right-justified within a field of size E2. However, if I2 is less than or equal to the length of E1 the value of E1 is returned unchanged. When 3 arguments are given the numeric interpretation of the first argument (N1) is formed, this value is then rounded to have I3 decimal places after the decimal point (including possible trailing zeros) and the result is right-justified in a field of I2 spaces.	$J(39,3) $J(39,4,1) $J(-5/3,0,0)	" 39" "39.0" "-2"
$LENGTH(E1)	$LENGTH returns the number of characters in string E1; that is, it returns the string length.	$L("ABC") $L(X) $L(X)	3 4 where X="ABCD". 0 where X=" ".
$NEXT(VN)	$NEXT returns the next numerically lowest existing subscript above that specified in VN. If there is no higher subscript, the result returned is -1. Note that VN need not exist.	$N(A(1)) $N(A(-1)) $N(^MUG(4))	2 if A(2) is defined (either pointer or datum). -1 if no subscripts exist. 6 if ^MUG(6) exists and ^MUG(5) does not.
$PIECE(E1,E2,I3) $PIECE(E1,E2,I3,I4)	$PIECE inspects a specified string (E1) and takes from it that substring which lies between two specified occurrences of a particular dividing string (E2). The resulting substring begins immediately after the I3 minus one (cont'd.)	$P("A.BX.Y",".",2) $P("76A666","6",1,4) $P(ST,".",2) $P(ST,".",1)	"BX" "76A66" " " if ST does not contain a period. ST if ST does not contain a period.

Table of Functions (concluded)

Function Name and Syntax	Explanation	Examples	Results
	occurrence of the dividing string and ends immediately before the I4th occurrence (or I3rd if I4 is unspecified).		
$RANDOM(I1)	$RANDOM returns an integer in the range 0 through I1 minus one. Each integer value has equal probability of occurrence.	$R(2) $R(100)	0 or 1. An integer between 0 and 99.
$SELECT(T1:E1,T2:E2, . . ., Tn:En)	$SELECT evaluates the left-hand expression in each of a series of expression pairs, one at a time in left-to-right order, until one is found with a truth value of 1 (true), then returns the value of the right-hand expression of that pair. In each pair, the truth-valued expression (Tn) on the left is separated from its "result" expression on the right (En) by a colon. There may be one or more pairs in the series, but at least one of the truth-valued expressions must be true.	$S(A=3:5,1:0) $S(X=7:"HI",1:"BYE") $S(Z=1:1,Z < 0:0,1:2)	5 if A has the value of 3. "BYE" if X has a value not equal to 7. 0 if X is negative.
$TEXT(L1) $TEXT(L1+I2) $TEXT(+I1)	$TEXT returns the text of a specified line of the current routine, including the line-start indicator and label. Lines may be referenced by label (L1), by a positive offset from a label (L1+I2), or by a positive offset from the beginning of the routine (+I1). If the specified line does not exist, the empty string is returned.	$T(LINE) $T(A+3) $T(+2)	Returns the text of the line labeled LINE. Returns the text of the third line after the line labeled A. Returns the second line of the routine.
$VIEW (argument syntax specified by the implementor)	$VIEW is an implementation-specific function available (at the option of the implementor) for providing implementation-specific data.	–	–
$Z_	Any additional function not included in the Standard should begin with "$Z".	–	–

$JOB	Each executing MUMPS process has its own unique job number, a positive integer, which is the value of $JOB.
$STORAGE	$STORAGE is the number of unused characters remaining in the user's partition.
$TEST	$TEST contains the last computed truth value. The value is set by the execution of the most recent IF command containing an argument, or by an OPEN, LOCK, or READ with a timeout.
$X	$X is the column or horizontal position at which the carriage (for a printing device) or the cursor (for a video terminal) lies for the current device. The first column is defined as column 0, the second column as column 1, and so on. Therefore, $X is 0 at the start of a line. After the first character in a line has been written, $X is 1. $X is initialized to zero by carriage return, and incremented by 1 for graphics.
$Y	$Y is the vertical position on the current device. The first line is defined as line 0, the second as line 1, and so on. $Y is initialized to zero by a form feed, and incremented by 1 for each line feed.
$Z_	Each nonstandard special variable should begin with "$Z".

FORMAT CONTROL

The following characters are used for format control during data display:

? specifies tab (right shift) to the column specified by the integer value of the expression following the question mark. For example, ?X tabs to column X. Remember (see $X above) that MUMPS regards the first column as column 0, the second as column 1, and so on. If X $X no shift takes place.

! specifies carriage-return and line-feed, or other similar new-line operation.

specifies a form-feed or new page, or similar operation.

LINES AND ROUTINES

A routine in MUMPS consists of an ordered series of lines. Lines are NOT free-format. Each consists of a line-start indicator (specified by the implementor), preceded by an optional line label, followed by zero or more commands and associated arguments, followed by an optional comment. Multiple commands in a line are separated from each other by exactly one

space, except that argumentless commands are separated from subsequent commands by exactly two spaces (where one blank indicates a null argument). Comments may be inserted into a line by placing a semicolon after the last argument or argumentless command. The remainder of the line is treated as a comment and is not executed.

A line label may begin with an alphabetic or with the "%" character. The subsequent characters may be any of the 26 alphabetics or any of the ten numeric digits. (To promote program portability alphabetic characters in labels should be upper-case only). Alternatively, the label may be entirely numeric digits characters (i.e., an integer). The label may be any length, but only the first eight characters are distinguished. Labels should be unique within a routine.

[For reasons of program portability, the length of a line of stored MUMPS code is currently limited to 255 characters, including the label, line start indicator, code and comments. There is no explicit limit on the number of lines in a MUMPS routine, but the routine including all local variables and temporary result storage should not exceed 4,000 characters.]

Note that the grouping of multiple commands into a single line is more than simple convenience. Commands such as FOR, IF, and ELSE treat the line as a meaningful unit.